

MMAV User's Guide

Version 2.0

DENALI SOFTWARE, INC.
1000 Hamlin Court,
Sunnyvale, CA 94089
Tel: (408) 743-4200
Fax: (408) 743-4209



info@denali.com
sales@denali.com
www.denali.com/support
www.ememory.com

All rights reserved

Confidentiality Notice

Denali Software, Inc. Sunnyvale, CA 94089

© 2006 Denali Software, Inc. All rights reserved.

Release: July 7, 2008

No part of this information product may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise without prior written permission from Denali. Information in this product is subject to change without notice and does not represent a commitment on the part of Denali.

The information contained herein is the proprietary and confidential information of Denali or its licensors, and is supplied subject to, and may be used only by Denali's customers in accordance with, a written agreement between Denali and its customers. Except as may be explicitly set forth in such agreement, Denali does not make, and expressly disclaims, any representations or warranties as to the completeness, accuracy, or usefulness of the information contained in this document. Denali does not warrant that use of such information will not infringe any third party rights, nor does Denali assume any liability for damages or costs of any kind that may result from use of such information.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraphs (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

Destination Control Statement

All technical data contained in this product is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

1	PREFACE	7
1.1	Audience Prerequisites	7
1.2	Typographical Conventions	7
1.3	Getting Help	8
1.3.1	Product Documentation	8
1.3.2	Related Information	8
1.3.3	Contacting Tech Support	9
1.3.4	Training Courses	9
1.4	How to Use This Guide	9
2	USING THE PUREVIEW GRAPHICAL TOOL	11
2.1	Launching PureView	11
2.2	Creating/Editing SOMA File	12
2.2.1	Denali SOMA files	12
2.2.2	Obtaining SOMA Files from eMemory.com	14
2.3	Using the PureView GUI	17
2.3.1	Viewing SOMA Files in PureView	17
2.3.2	PureView “File” Pull-down Menu	19
2.3.3	PureView “Options” Pull-down Menu	21
2.4	Creating an HDL Shell with the PureView GUI	23
2.5	Creating or Modifying a SOMA file	24
2.5.1	Creating an HDL shell via Command Line	27
3	DEBUGGING MEMORY USING PUREVIEW	30
3.1	Denali Memory Database Files	30
3.2	Interactive Debugging During Simulation	31
3.2.1	UNIX Shell Invocation	31
3.2.2	Simulator/Testbench Invocation	32
3.3	Opening up the Simulation Results File	32
3.4	Selecting Memory Instances	33
3.5	PureView Debugging Windows	34
3.5.1	Memory Contents Window	34
3.5.2	Memory Contents Transaction Summary	36
3.5.3	Transaction History View	36
3.6	Post-Processing with PureView	37
3.7	Using PureView with Mentor Graphic’s Seamless HW/SW Co-Verification	37
4	USING DENALI’S MEMORY MODELER ADVANCED VERIFICATION	39
4.1	Controlling Your Memory Simulation Models - The .denalirc File	39
4.1.1	Register File Specific .denalirc Parameters	47
4.1.2	IBM-EDRAM Specific .denalirc Parameters	47
4.1.3	RDRAM Specific .denalirc Parameters	47
4.1.4	RLDRAM Specific .denalirc Parameters	48
4.1.5	DDR-II SDRAM Specific .denalirc Parameters	48
4.1.6	DDR-II and DDR3 Specific .denalirc Parameters	49
4.1.7	ESSRAM Specific .denalirc Parameters	49

4.1.8	Mentor Graphics ModelSim Specific .denalirc Parameters	49
4.1.9	Mentor Graphics Seamless HW/SW Co-Verification Specific .denalirc Parameters	50
4.1.10	OneNand Flash .denalirc Parameters	50
4.1.11	.denalirc Summary Table	52
4.2	Setting .denalirc Options Dynamically During Simulation	54
4.3	Licensing Solutions	55
4.3.1	Simulator Queuing for Denali Licenses	55
4.3.2	Speeding up License Checkout	55
4.4	The Denali Tcl Interface	56
4.4.1	Using with ModelSim	56
4.4.2	Using with NCSIM and other Tcl Interpreters	56
4.4.3	Tcl Commands	56
4.4.4	Callback Commands	59
4.5	Initializing Denali Memories	61
4.5.1	Memory Address Determination	61
4.5.2	Initial Contents of Memories	61
4.5.3	Loading Memories From a File	63
4.5.4	Memory Content File Format	64
4.6	Specifying Memory Instances	66
4.7	Resetting the Memory Contents	67
4.8	Reading and Writing Memories	68
4.8.1	Masked Memory Writes	70
4.9	Saving and Comparing Memory Contents	70
4.10	Recalculating Clock Cycle Time	72
4.11	Re-loading SOMA Files and Changing Timing Parameters on-the-fly During Simulation .	72
4.12	Error Message Control	75
4.13	Forcing Clock Cycle Recalculation	76
4.14	RDRAM (Rambus) Specific Model Considerations	76
4.14.1	Turbo Channel Model for RAMBUS	76
5	MMAV SPECIAL VERIFICATION FEATURES	81
5.1	Setting Assertions on Memory Transactions	82
5.1.1	Memory Access Assertions	82
5.1.2	Data Access Assertions	83
5.1.3	Global Memory Access Assertions	85
5.2	Parity Checking Assertions	86
5.3	Dynamically Enabling and Disabling Assertions	86
5.4	Error Injection Routines	87
5.4.1	Error Injection	87
5.4.2	Fault Modeling	90
5.5	Logical Addressing with MMAV (Method #1)	92
5.5.1	XML Basics	92
5.5.2	Depth, Width Expansion	93
5.5.3	Interleaving	95
5.5.4	Address Scrambling	96

5.5.5	Data Bit Reordering and Masking	96
5.5.6	Creating Holes	97
5.5.7	Putting it all Together	97
5.5.8	Interfacing to MMAV	100
5.6	Logical Addressing (Method #2)	100
5.7	Address Scrambling	105
5.8	Scratchpad Memories	106
5.9	Verilog Callbacks (new in 3.2)	108
5.9.1	Callback Interface	108
5.9.2	Callback Initialization	108
5.9.3	Callback Handling	109
5.9.4	denaliMemCallback Registers	110
5.9.5	denaliMemCallback Tasks and Functions	111
5.10	Using MMAV with Mentor Graphic's Seamless HW/SW Co-Verification Product	113
5.11	Using MMAV for Embedded ASIC Memories	114
5.11.1	Register Files	114
5.11.2	Embedded SRAM	115
5.11.3	Embedded DRAM	117
6	MMAV TESTBENCH INTEGRATION	118
6.1	Verilog Interface	118
6.1.1	Simulating with MMAV and Verilog	118
6.2	VHDL Interface	123
6.2.1	Simulating with MMAV and VHDL	123
6.3	SystemC Interface	126
6.3.1	MMAV and SystemC Overview	126
6.3.2	Simulating with MMAV and SystemC	126
6.4	Specman Interface	128
6.4.1	MMAV and Specman Overview	128
6.4.2	Simulating with MMAV and Specman	129
6.4.3	Configuration Register and Memory Access	131
6.4.4	Extending sn_denali_unit to include all MMAV Functions as Methods	131
6.4.5	Viewing Memory Transactions in Waveforms	133
6.4.6	Example Testcase	133
6.5	Vera Interface	134
6.5.1	MMAV and Vera Overview	134
6.5.2	Simulating with MMAV and Vera	135
6.5.3	Initializing Denali Memory Models from Vera Testbench	136
6.5.4	Processing Callbacks	137
6.5.5	Using other Denali functions from Vera Testbench	139
6.5.6	Example Testcase	140
6.6	NTB Interface	141
6.6.1	MMAV and NTB Overview	141
6.6.2	Simulating with MMAV and NTB	141
6.6.3	Instance and Transaction Classes	141

6.6.4	Processing Callbacks	163
6.6.5	Example Testcase.	163
6.7	SystemVerilog Interface	166
6.7.1	MMAV and SystemVerilog Overview	166
6.7.2	Simulating with MMAV and SystemVerilog	166
6.7.3	Configuration Register and Memory Access.	168
A	GETTING TECHNICAL SUPPORT	196
A.1	The Denali History File	197
A.1.1	Understanding the Denali History Files (HistoryFile in .denalirc):.	198
A.1.2	HistoryDebug Mode (HistoryFile AND HistoryDebug in .denalirc):.	198
A.2	Understanding the History File	200
A.2.1	SIM READ Entry.	200
A.2.2	MASKED SIM Write Entry.	200
A.2.3	Debug Read	200
A.2.4	Debug Write.	200
A.2.5	File Load	200

1 Preface

Welcome to the *MMAV User's Guide*. This manual describes use of the Denali Memory Modeler Advanced Verification (MMAV) and PureView software.

1.1 Audience Prerequisites

This guide, and the product it describes, are intended for chip designers and verification engineers. Readers of this guide should have a solid understanding of Verilog and VHDL, and Tcl.

1.2 Typographical Conventions

This guide uses the following typographical conventions:

- Literal string values for commands, filenames, and command-line options are shown in monospace.

Example: The `.denalirc` file allows you to change runtime modeling parameters.

NOTE: *Literals such as register names are shown in regular font.*

- Variables where you should substitute a context-specific value are shown in angle brackets.

Example: To search for a file, type `grep <filename>` where `<filename>` is the name of your file.

- UNIX environment variables are shown using the standard notation.

Example: The `$DENALI` environment variable specifies your product installation home directory.

- For options where the user can choose zero or more from a series of options, the syntax is shown using angle brackets and pipes

`myscript [-option1 <value>|option2]`

So to run the fictitious `myscript` specifying a machine called `server1` in debug mode:

`myscript -server server1 -debug`

- Syntax examples are shown in a text box.

```
% verilog [all of your regular arguments] \
$DENALI/ddvapi/verilog/denaliPcie.v \
+incdir+$DENALI/ddvapi/verilog
```

1.3 Getting Help

You can find help, training, and online documentation as described in the following sections.

1.3.1 Product Documentation

If you have any questions about using Denali products, consult the product documentation that is installed on your network or found in the Denali software release structure.

`$DENALI/docs/`

where `$DENALI` is the Denali home directory.

1.3.2 Related Information

You can access the following related information:

- ***Getting Started User's Guide*** that provides details on how to get started with Denali VIP products. This includes details on downloading the software, creating an installation directory, un-compressing the installation package, installing the software license, and a list of supported tools and platforms.

You can find `GettingStartedUserGuide.pdf` at `$DENALI/docs/`

- FAQs for Denali products on the Web at the following URL:

<http://www.denali.com/support>

- SOMA file from:

<http://www.ememory.com>

- MMAV releases from:

<http://www.denali.com/support>

For more information about Denali and its products, check out:

<http://www.denali.com>

1.3.3 Contacting Tech Support

All users from organizations that have purchased a valid software license are eligible to receive technical support. You can contact Denali's support center as follows:

For email support:

- Go to www.denali.com/support
- On the left-hand side, choose **Support > Product Support > Create New Ticket**.
- Open a support ticket with detailed information of problem along with a tracefile of simulation with following .denalirc configuration options set.

```
Historyfile denali.his
Historydebug on
Tracefile denali.trc
```

(For more information on the .denalirc file and trace file settings, see See “Controlling Your Memory Simulation Models - The .denalirc File” on page 39..)

- Telephone your local support center:
 - United States : 408-743-4200, option 3
 - Europe : +44-1494-481030, europe@denali.com
 - Japan : +81-3-3511-2460, japan@denali.com

1.3.4 Training Courses

Denali offers a complete suite of training courses for its products, including PureSpec. Contact your local technical support center for information on current course offerings.

1.4 How to Use This Guide

This guide is divided into the following chapters:

- [Chapter 1: “Preface”](#) provides an introduction to the manual itself.
- [Chapter 2: “Using the PureView Graphical Tool”](#) describes both the SOMA parameters viewing and editing capabilities of the new PureView GUI.
- [Chapter 3: “Debugging Memory Using PureView”](#) provides information about the PureView debugger with a unique graphical view of your memory contents during simulation.
- [Chapter 4: “Using Denali’s Memory Modeler Advanced Verification”](#) provides details on how to use the MMAV product that extends Denali’s world class memory modeling capability with advanced verification features aimed at enhancing your memory sub-system verification.

- [Chapter 5: “MMAV Special Verification Features”](#) lists a suite of additional verification functions that allow you to add verification checks to your testbenches and also organize physical memories into logical memory views.
- [Chapter 6: “MMAV Testbench Integration”](#) lists and describes the various supported testbench interfaces.
- [Appendix A: “Getting Technical Support”](#) describes various methods to get the technical support from Denali.

2 Using the PureView Graphical Tool

PureView graphical user interface enables you to configure a SOMA file and combines all the previous capabilities of Memory Maker with the enhanced debugging capabilities of PureView. This reduces the number of desktop windows required for Denali products.

This chapter describes both the SOMA viewing and editing capabilities along with the debugging features of the new PureView GUI.

2.1 Launching PureView

The PureView GUI is launched from your Unix or Linux shell command line by typing:

```
shell > $DENALI/bin/pureview &
```

This will bring up the following graphical user interface (GUI):

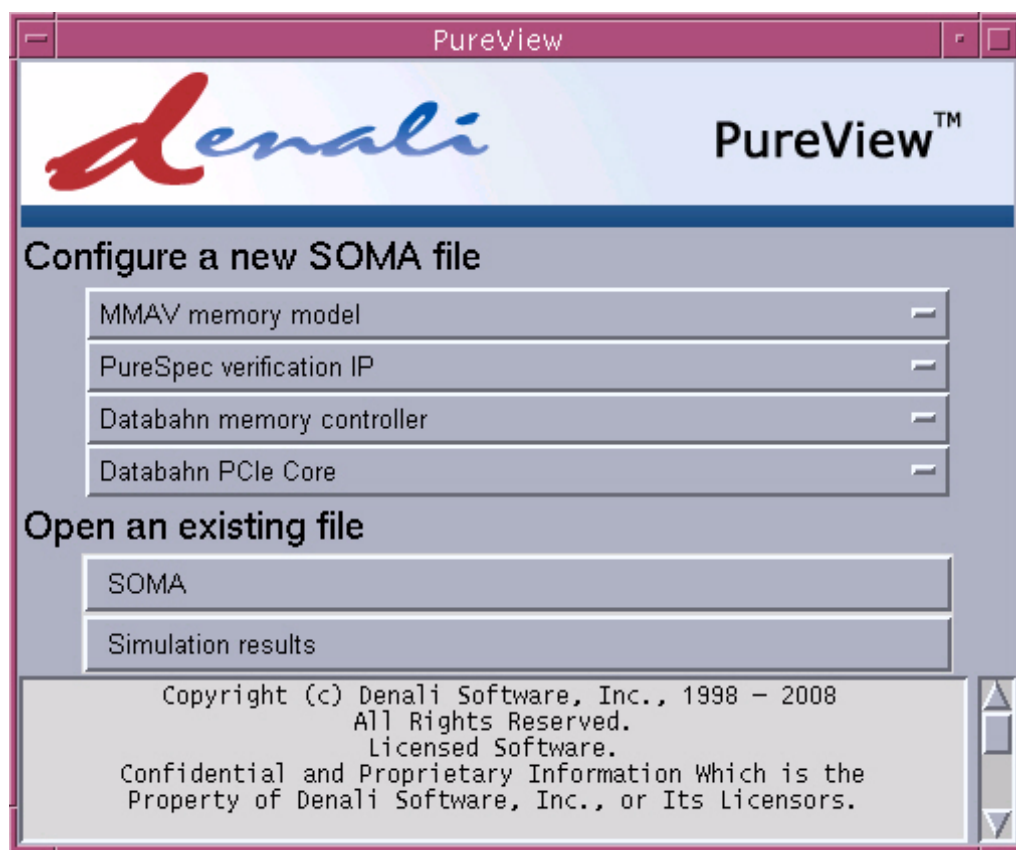


FIGURE 2-1: PureView GUI

2.2 Creating/Editing SOMA File

PureView is used to create new SOMA (Specification of Memory Architecture) files, view and edit the parameters of a Denali or memory vendor created SOMA file, and to generate an HDL wrapper to be used in your simulation. Before discussing these features, an overview of Denali's SOMA file technology is needed.

2.2.1 Denali SOMA files

The core of Denali's Memory Modeling solution are class based 'C' models for all popular memory class architectures. These 'C' models cover all the features found across the entire memory class of a specific family of devices. For example, Denali has 'C' models for the following memory classes:

SRAM	DRAM	Non-Volatile	Volatile	Embedded ASIC/ FPGA
Synch-SRAM	DRAM	Flash	Card Memories	DRAM
Asynch-SRAM	EDO-DRAM	Synch-FLASH	MultiMedia Card*	SRAM

SRAM	DRAM	Non-Volatile	Volatile	Embedded ASIC/ FPGA
DDR SRAM	SDRAM	EEPROM	MemoryStick*	FLASH
QDR SRAM	SGRAM	SEEPROM	MemoryStick Pro*	Register Files/Arrays
QDR-2 SRAM	Enhanced-SDRAM	PROM	SecureDigital **	
SigmaRAM	DDR-SGRAM	SMROM	SecureDigital IO**	
FIFO	DDR/DDR2-SDRAM	Compact Flash	* = For Licensee only	
Cellular RAM	FCRAM/FCRAM2	Atmel Serial/Parallel/SPI Flash	** = For SDA Members only	
Mobile RAM	RDRAM	AND/NAND/NOR Flash		
	RLDRAM	Sharp Flash		
	RLDRAM2	SST Serial Flash		
	GDDR2	One NAND		
	GDDR3	DDR-NVM		
	GDDR4	LPDDR2-NVM		
	GDDR5			
	DDR3			

Because there are numerous variations of the above memory classes from different memory vendors, Denali uses SOMA files to describe the specific memory features and timing for a specific part number. These SOMA files uniquely parameterize our highly optimized 'C' core models to adjust their behavior and timing according to the SOMA specification.

SOMA files are widely used by memory vendors to specify the behavior of memory components. These files can be freely downloaded from a number of memory vendor web sites. In partnership with these vendors, Denali also maintains a repository for SOMA files on the Denali eMemory website (<http://www.eMemory.com>). If you are unable to locate a SOMA file for a device you are interested in, it is our policy to deliver SOMA files upon request. In general, you can exercise the following steps for obtaining SOMA files:

1. Search Denali's eMemory website (<http://www.eMemory.com>)
2. Search the memory vendors website for the pertinent SOMA file
3. Modify an existing SOMA file for a similar device
4. Request the SOMA file from <http://www.eMemory.com> (be sure to include references to any pertinent device specifications and datasheets)

2.2.2 Obtaining SOMA Files from eMemory.com

Denali SOMA files completely characterize our C-based memory models for a specific vendor and speed grade. Denali meticulously creates and verifies these files using the vendor datasheets and our expertise in memory devices.

SOMA files for specific memory vendor part numbers can be obtained from our eMemory website. The URL is: <http://www.eMemory.com>. The first time you access this page, you will be asked to register for downloads. This ensures that if one of your SOMA files is ever updated, you will receive notice of the event allowing you to go back and download the latest version.

MemCon 2008
MEMCON08
MemCon San Jose 2008 is the largest worldwide conference and exhibition addressing the technology, business, and system design strategies for memory and storage.
[Register Now](#)

Denali PureSpec™
PureSpec is the industry's most trusted verification IP. PureSpec includes a configurable BFM, protocol monitor, and complete assertion library for all standard interfaces:
PCI Express
Advanced Switching
USB 2.0, OTG
Serial ATA
CE-ATA
Ethernet
AMBA, AXI

Denali User Registration

Thank you for registering at Denali.com, we are proud to offer the most comprehensive and high-quality resources for designing and verifying standard chip interfaces. Registration provides you with immediate access to market reports and a host of resources for design and verification with semiconductor memory and other standard interfaces including: PCI Express, USB, SATA, and more.

Resources include market reports, webcasts, whitepapers, EDA and IP solutions, and a searchable online database of information for over 10,000 semiconductor memory devices.

Complete the form to automatically receive your password for instant access!

Registration Form

* Indicates a required field

*Email address:

*First name:

*Last name:

*Job title:

*Company:

Postal address:

FIGURE 2-2: eMemory.com Registration Page

Once you are a registered eMemory.com user, you can search for SOMA files for your design.

The screenshot shows the Denali eMemory.com website. The top navigation bar includes links for 'Products & Solutions', 'Support', 'Partners', 'News', 'Events', and 'Company'. A search bar is located in the top right corner. On the left side, there is a sidebar with 'eMemory' and 'Denali Customers' sections. The main content area is titled 'Memory Search' and contains a search form. The form has dropdown menus for 'Vendor' (set to '-- All Vendors --'), 'Class' (set to '-- All Classes --'), and 'Data Width' (set to '-- All Widths --'). There are also input fields for 'Size' with 'min' (0Mb) and 'max' (1Tb) values, and a 'Part #' field. A 'Search' button is at the bottom right of the form. To the right of the form, a text box explains the search engine's substring matching and wildcard usage, and includes a link to 'request a SOMA'.

FIGURE 2-3: eMemory.com SOMA Search Page

You can search for the specific memory simply specifying either **Vendor**, **Class**, **Data Width**, **Size**, or **Part Number**.

You must limit your memory search to a specific Vendor, Class, Data Width, Size and/or Part Number. The search engine performs a substring match on your input to the "Part #" field, which also accepts * as a wildcard.

You can then select the relevant SOMA and **Add to cart**. This displays the **Download Cart**. Using this interface, you can either **Send cart now** to get the SOMA files emailed to you or **Remove Selected** or **Remove All** SOMA files.

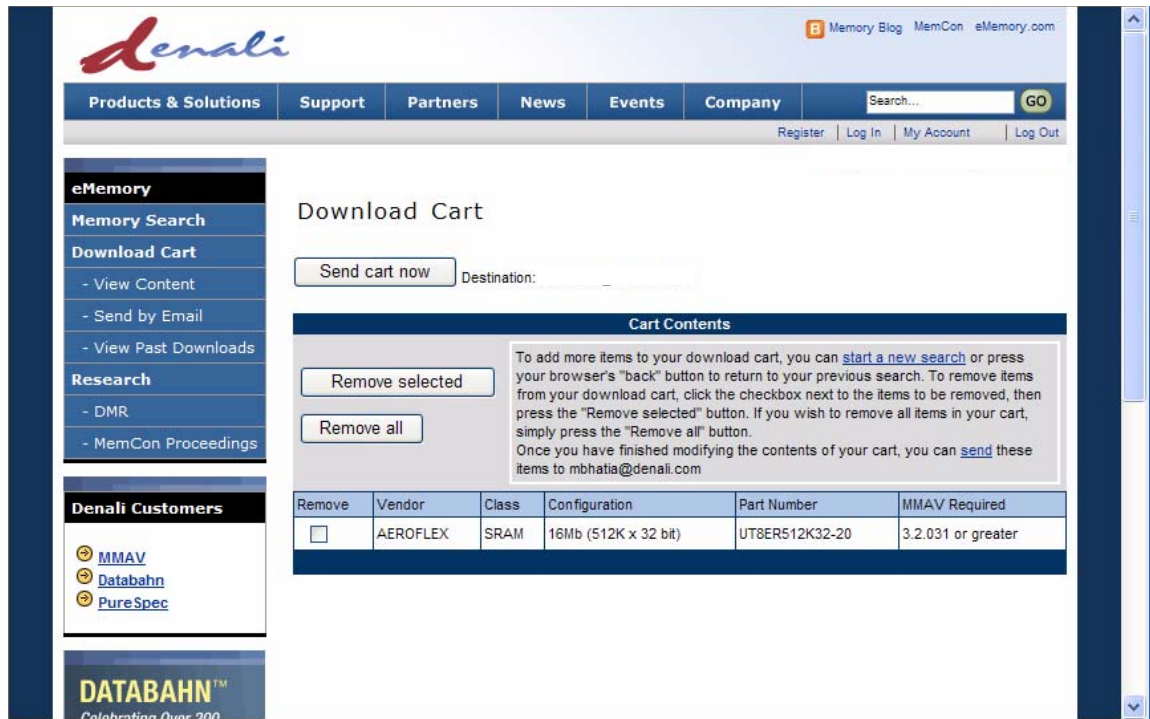


FIGURE 2-4: eMemory.com Download Cart

Once your files have been emailed, you can now take a look at the settings using Pure-View. For details, refer to “Using the PureView GUI” on page 17.

NOTE: *If you are unable to find the part you are looking for, you can request a SOMA using the following interface.*

FIGURE 2-5: SOMA Request

2.3 Using the PureView GUI

To start PureView, enter the following command from the UNIX shell:

```
$DENALI/bin/pureview &
```

This will bring up the PureView window where you can choose to either create a new SOMA specification file, or open an existing SOMA file. A SOMA file contains all the necessary functional and timing specifications to model any memory device.

2.3.1 Viewing SOMA Files in PureView

To read in an existing SOMA file, select the “Open SOMA file” button on the main PureView window. SOMA files can have either an XML *.soma* suffix or a text file *.spc* file suffix. Select a SOMA file in the current directory (or traverse to another directory) and click on the “Open” button on the file selection window. If the SOMA file is not in the current directory, use the directory browsing features at the top of the file selection window.

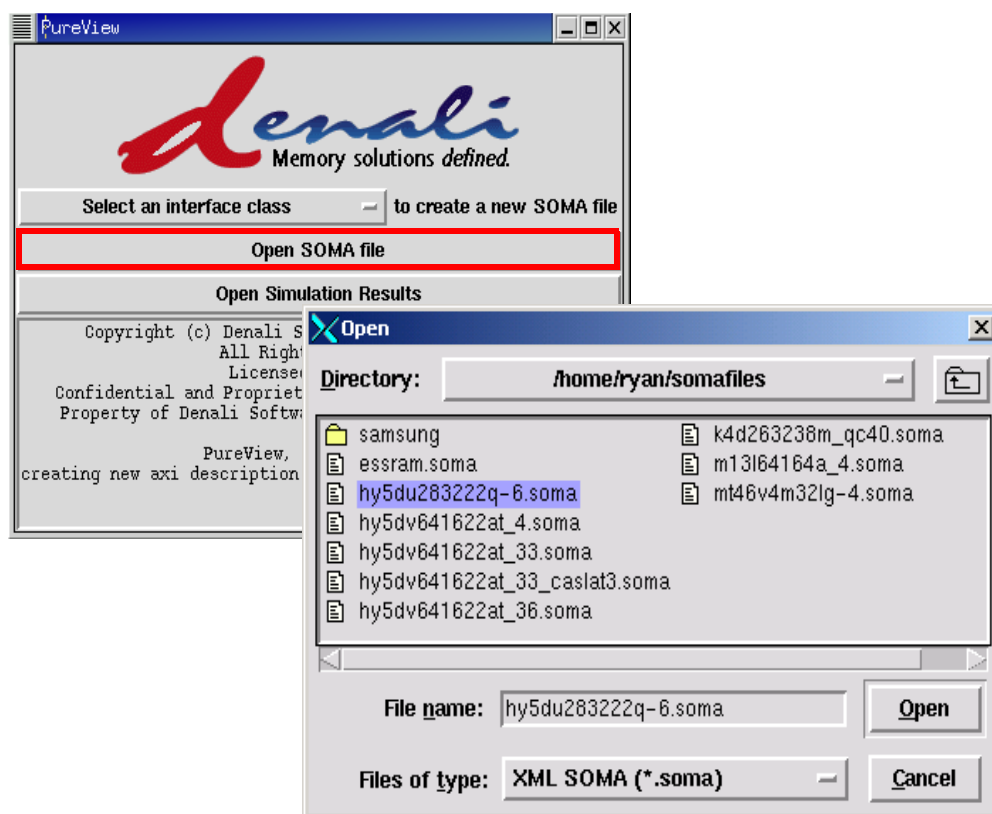


FIGURE 2-6: PureView “Open” SOMA File Window

2.3.2 PureView “File” Pull-down Menu

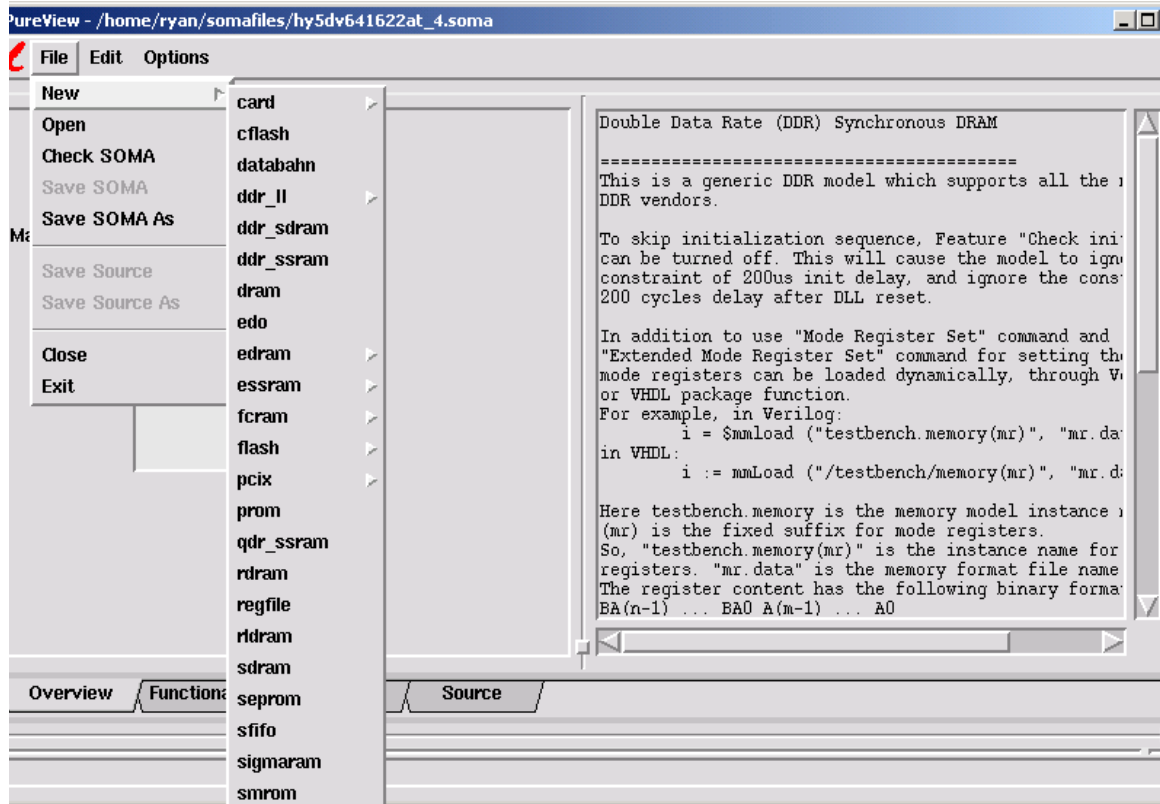


FIGURE 2-7: PureView “File” Menu

“New” SOMA File

This option is used to create your own SOMA file using Denali’s default SOMA file for a specific memory class.

“Open” a SOMA File

This option allows you to open up a different SOMA file. It will bring up a dialog box allowing you to choose another SOMA file to open up in PureView.

“Check” a SOMA File for Errors

This option will check the SOMA file for syntax errors and for illegal feature combinations. If creating your own SOMA file from scratch, it is recommended that you check your SOMA file before saving it and using it in a simulation. NOTE: this check will not ensure the correctness of your SOMA file from a timing and functionality standpoint, it will only check for illegal parameters set in the SOMA file.

Note that the **Save SOMA** or **Save SOMA As...** commands discussed below cause the same checks as the Check SOMA command, and gives you the option of cancelling the save or proceeding with the issues intact.

“Save SOMA” File Option

This option will allow you to save your SOMA file. The Save SOMA option is available once any changes to a SOMA file are detected. If you are simply viewing a Denali created SOMA file, you will not need to re-save the SOMA file, unless you have made changes to the parameters.

“Save SOMA As” File Option

This option allows you to save a newly created SOMA file or to save an existing SOMA file as a different filename.

“Save Source” Option

This option allows you to save the HDL wrapper file generated in Figure 2.4, “Creating an HDL Shell with the PureView GUI,” on page 23 to disk. This option will use the SOMA file name and use either a **.vhdl** or **.v** file extension (for VHDL and Verilog sources respectively) by default.

“Save Source As” Option

This option allows you to save the HDL wrapper file generated in Figure 2.4, “Creating an HDL Shell with the PureView GUI,” on page 23 to disk. This option will use the memory model class name and use either a **.vhdl** or **.v** file extension (for VHDL and Verilog sources respectively) by default. New in 3.2, you can save off an HTML datasheet of the interface you are modeling.

“Close” Option

This option closes the current PureView window, but does NOT close the entire PureView application

“Exit” Option

This option closes the current PureView window as well as any other open PureView windows and exits PureView.

2.3.3 PureView “Options” Pull-down Menu

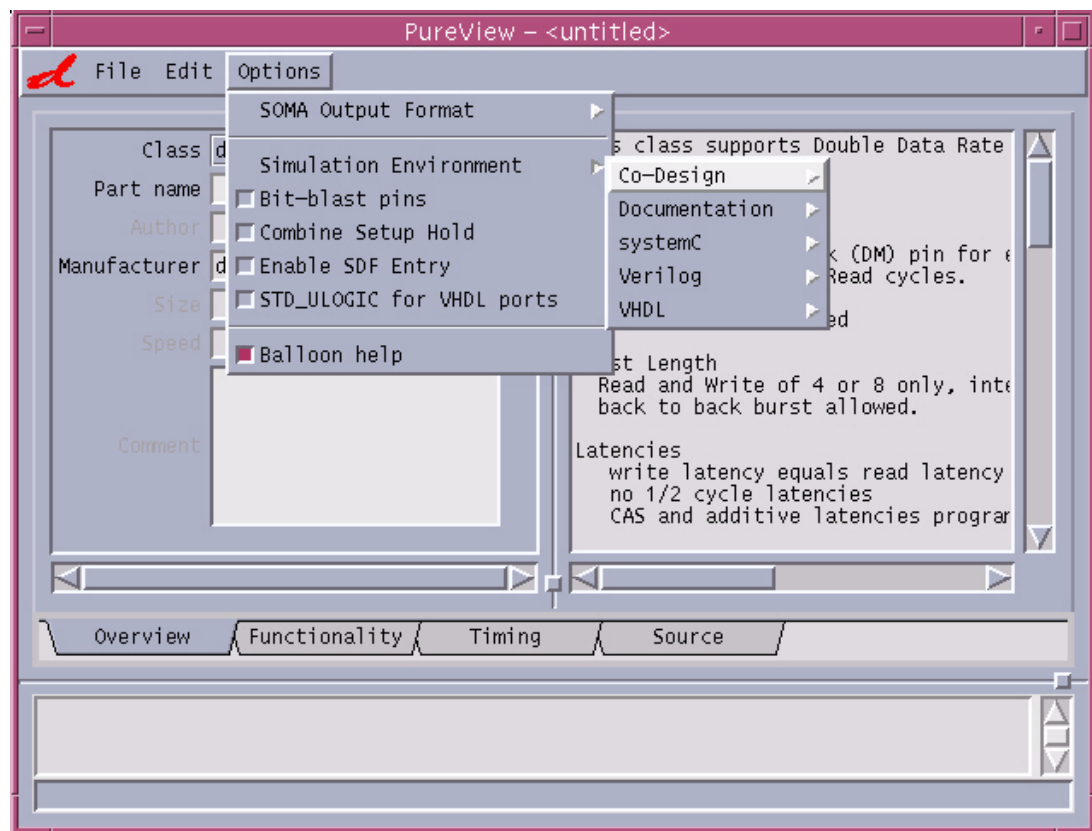


FIGURE 2-8: PureView “Options” Menu

SOMA Output Format

This option gives you the ability to save the SOMA file out in any of three supported formats. By default, all SOMA files downloaded from eMemory.com are in a compressed **XML** format. You can save these as either **Uncompressed XML** format or as a text file (**Version 0.001**).

Simulation Environment

This option is used to select the simulation environment you are using for the HDL shell generation process. You can select from **Verilog**, **VHDL**, **HTML Datasheet**, and **SystemC**. Once selected, the PureView “**Source**” window will display the HDL shell file and you can use the **File -> Save Source** menus to save the HDL shell for simulation.

The datasheet option lets you save off an HTML Datasheet of your interface that is being modeled. This allows you to quickly extract the features into any documentation. An example of this is shown below:

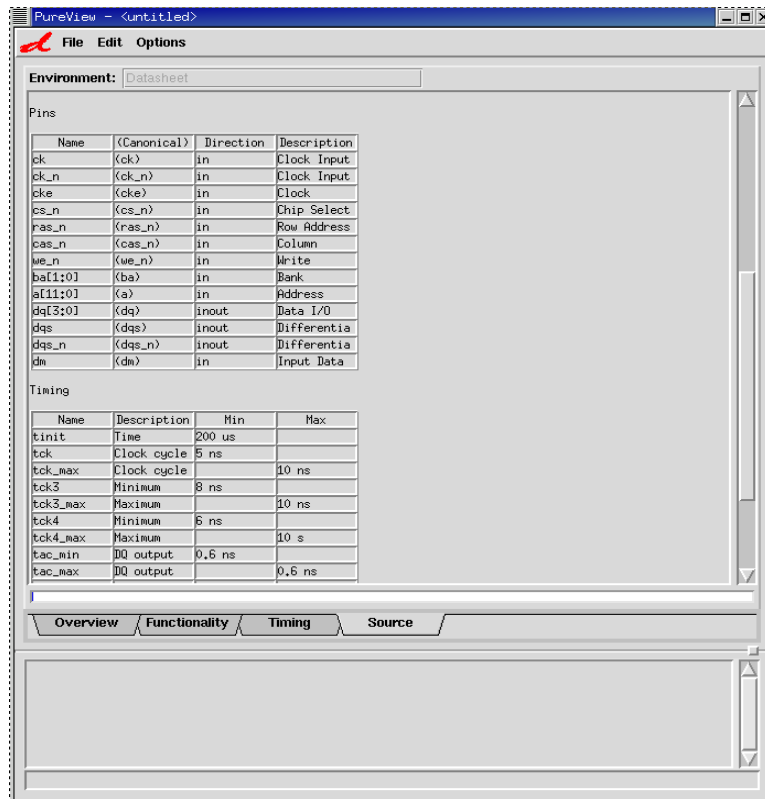


FIGURE 2-9: PureView “Datasheet” Source Window

Bit-blast Pins

This option will expand all data buses into individual single bit ports for the HDL shell file. Additionally, individual pins may be bit-blasted with the **“Blast Bits”** check boxes in the **“Pins”** (right) half of the **“Functionality”** tab. Refer to Figure 2-11, “PureView “Functionality” Window” for a sample window.

STD_ULOGIC for VHDL Ports

This option will convert all VHDL port signal types to STD_ULOGIC. The default signal type is STD_LOGIC.

Balloon help

This option turns on the “mouse over” help information. You can disable this by un-checking this box.

2.4 Creating an HDL Shell with the PureView GUI

Once you have successfully loaded a SOMA file into PureView, you now need to create an HDL shell of the memory suitable for your particular simulation environment. This is accomplished by selecting the “**Source**” tag in the bottom right hand corner of the PureView SOMA description window.

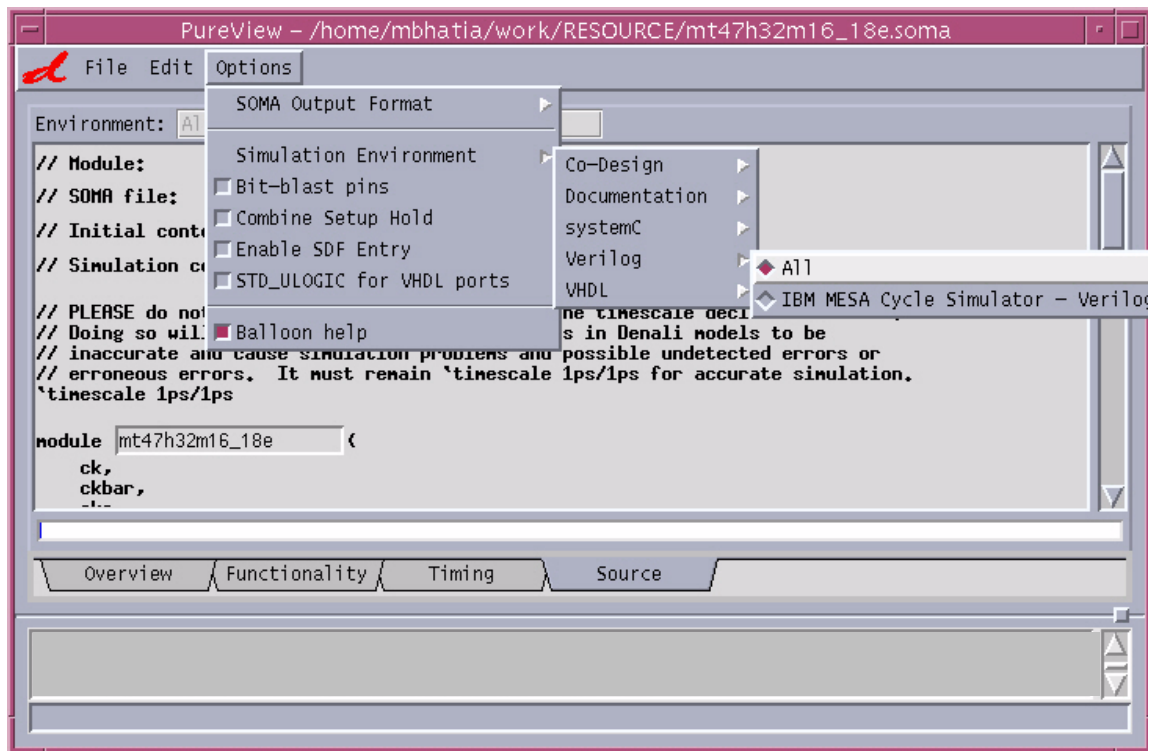


FIGURE 2-10: PureView HDL Source Window

The **Source** window will initially be blank. Select the **Options -> Simulation Environment** menu tag at the top of the window. From here select either Verilog or VHDL depending on the HDL language your simulator supports. Then proceed to select the actual simulator you will be using. The actual code for the HDL simulator will appear in the Source window. You will notice that some of the fields can be edited. These fields have been pre loaded with the appropriate data from the SOMA file you read and do not need to be modified. If you choose to modify the location of the *.spc* file or the initial contents file, you can select the “...” button next to these fields and a file selection box will pop up. Use this box to select an alternative file.

The HDL shell enables you to specify an initialization file (*init_file*). You can edit the path name to point to a file to be loaded at model initialization time. This file format is identical to the Memory Content File Format as described in [“Memory Content File Format” on page 64](#).

The last step is to write out this HDL shell to a file so it can be instantiated into your design. To do this, select the “File -> Save Source As” menu button at the top left of the Source window. A file selection box will pop up where you can name the HDL source file and write it to the appropriate directory. Once this is done, you have everything you need to start simulation with the particular memory device specified in the SOMA file.

NOTE: *New in 3.2, all of the verilog simulator options have been combined into a single source window. Because of the PLI standard interface, the Verilog HDL wrappers are now identical for all Verilog simulators. Select the “All” option for any Verilog simulator.*

2.5 Creating or Modifying a SOMA file

PureView allows you to create a new SOMA file from scratch as well as modify an existing SOMA file. This section will describe the process by which you can modify or create a SOMA file with PureView. Below are the steps you need to follow to create/modify a SOMA file.

1. Select an interface class OR read in an existing SOMA file

Once you have invoked PureView, either select “Open SOMA file” to modify an existing SOMA file or select “Select an interface class” from the main PureView window. A PureView class window will appear with 4 tabs on the bottom of it (Overview, Functionality, Timing, Source).

2. Select size, functional parameters, and pin names

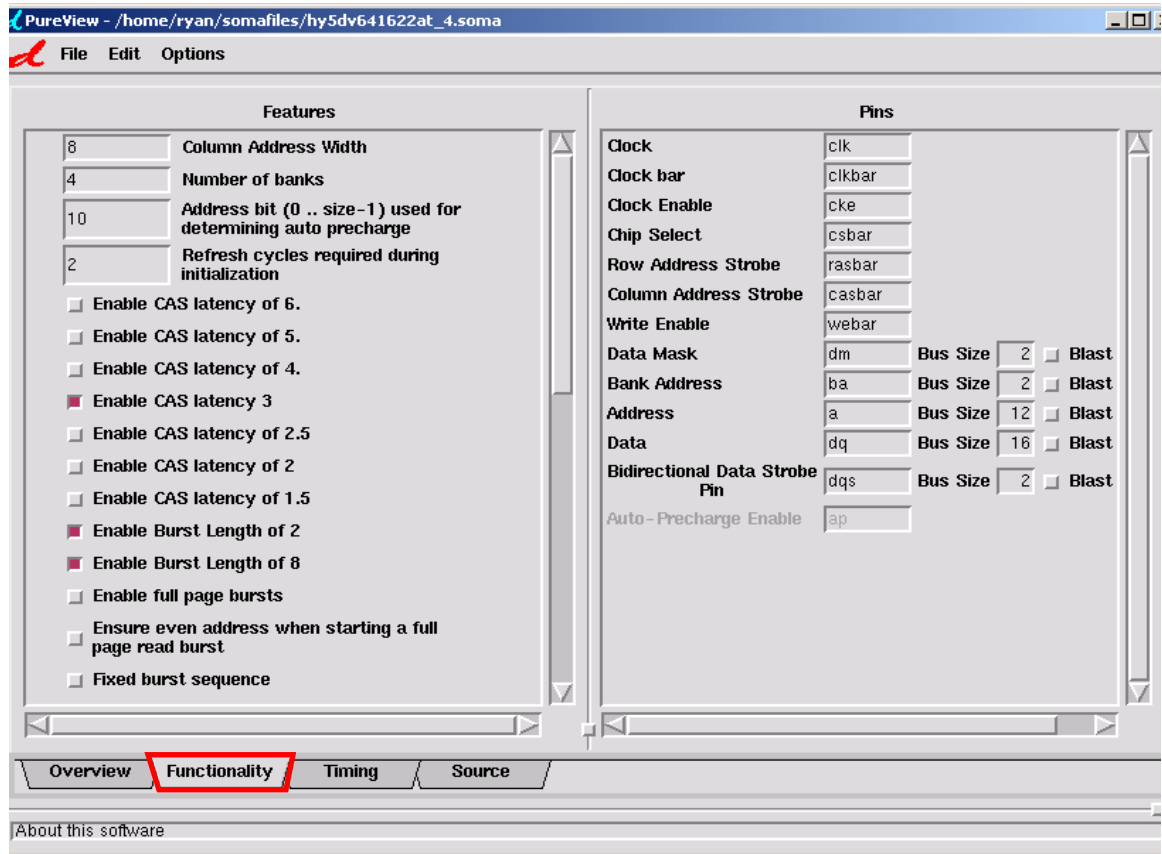


FIGURE 2-11: PureView “Functionality” Window

Select the **Functionality** Tab in the bottom of the PureView window. The functionality window will allow you to parameterize the simulation model for the class of memories you have selected. Each class has its own set of parameters that describe the size and functionality of the memory class. You can select and deselect these options in the left hand window that appears. The right hand window allows you to specify the pin names and widths for each pin type. The size of the device is determined by the pin widths of the address and data ports.

3. Pin Naming

Beginning with Denali version 2.800, pin naming was enhanced to provide you more control of the pin names generated in the HDL shell. Primarily enhanced for multi-port SRAM devices (embedded as well as external), these enhancements enables you to uniquely identify the naming of multiple ports. PureView now takes the following syntax in the Pins section: `name_prepend{port1_specifier, port2_specifier, port3_specifier}name_postpend`.

Example:

For a 2 port device and the desired HDL shell Output Enable pin names of: `OE_port1_N` and `OE_port2_N`, enter the Output Enable pin name in PureView as: **`OE_{port1,port2}_N`**.

4. Specify timing parameters

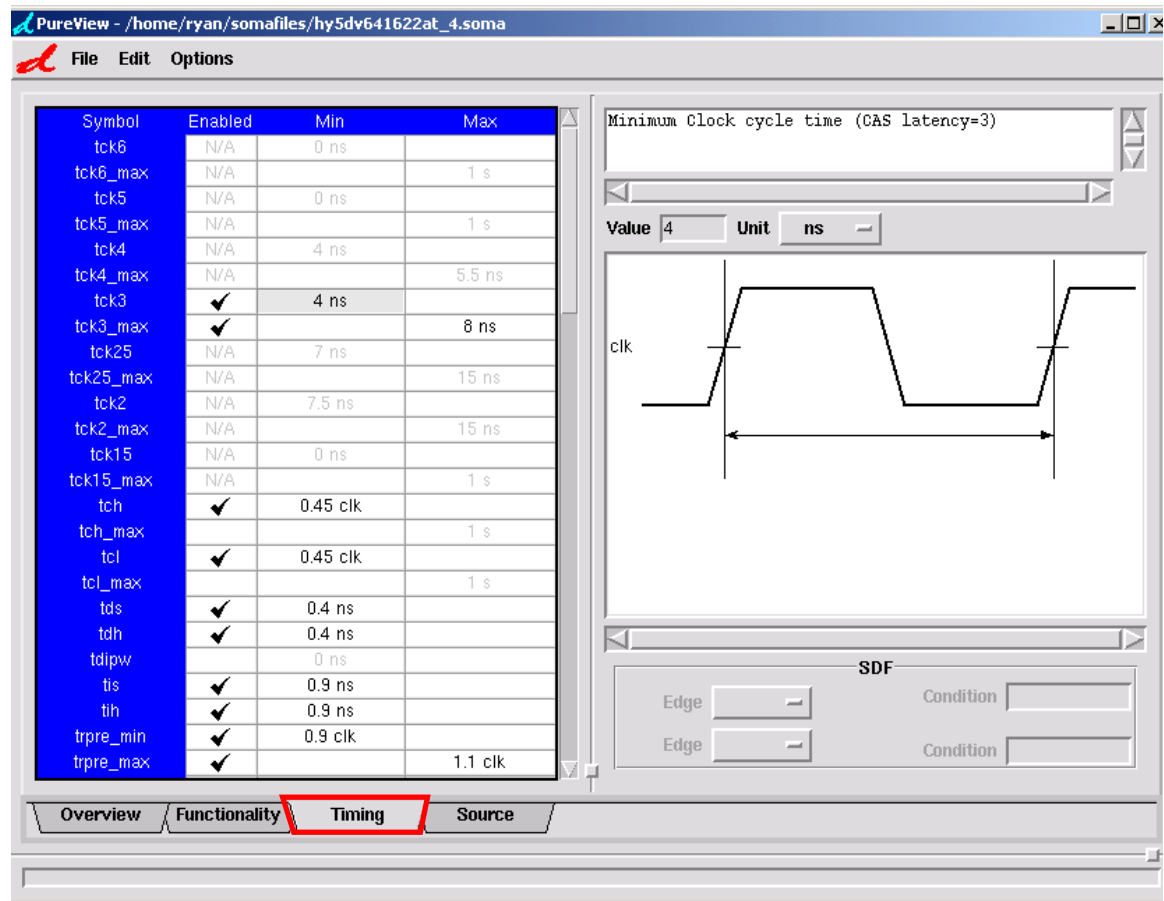


FIGURE 2-12: PureView “Timing” Window

Once you have selected the proper functionality and size of the memory, you now must specify the timing parameters relevant to this class and parameters you have set. To edit the timing parameters, simply left mouse button click on the timing parameter value. The description and value of this parameter will appear in the upper right hand corner of the window as well as a timing diagram that illustrates the timing parameter’s relationship. You can modify this timing parameter by right mouse clicking in the “Value” field and inputting the correct value. The magnitude of the value can be changed by clicking on the pull down menu to the right of the “Value” field. To change the value of the parameter, press RETURN while the cursor focus is in the “Value” input box.

You can enable or disable reporting of a particular error if the timing relationship specified is violated during simulation. To do this, simply click on the “check mark” to disable this check. Clicking on the blank spot under the “Enabled” column will turn the timing check back on. Any timing check that is not applicable to the parameters you set in the “Functionality” tab will be greyed out and cannot be edited.

5. Save the SOMA file

Once you are done setting all the functionality and timing parameters for the memory, you need to write out the SOMA file to disk. To do this, select the “File -> Save SOMA As” pull down menu at the top left hand corner of the window. A file selection dialog box will appear where you can select the name of the SOMA file. Once you have specified the file name for the SOMA file, click on “Save”. If you are simply modifying an existing SOMA file, you can select “File -> Save SOMA” and the original SOMA file will be overwritten by the new specifications you have input through the “Functionality” and “Timing” windows.

3. Write out HDL Shell

The last step is to write out the HDL shell for the memory you have just created or modified so you can add it to your simulation environment. To do this, follow the instructions outlined in Figure 2.4, “Creating an HDL Shell with the PureView GUI,” on page 23. **Caution:** Write out the HDL shell last. This will insure the HDL shell points to the correct SOMA file when you instantiate it in your design. If you write out the HDL shell before you save the SOMA file, you could be pointing to an outdated SOMA file.

2.5.1 Creating an HDL shell via Command Line

PureView is most commonly used to create the HDL shell necessary to instantiate in your simulation. This HDL shell contains the module definition for Verilog or the entity/architecture pair for VHDL for the device described in the SOMA file. The pins of the device are defined along with a call to the particular memory class object in “C”. This object is linked into the simulation via the denali.so shared library object provided in the Denali software. You can see a list of PureView options by typing:

```
pureview -batch -help
```

To create an HDL shell via the command line, the syntax is as follows:

```
pureview -batch -generate <target environment> <model name> -genoption  
option1,option2 -genoutput <HDL shell file name> <soma file>
```

Example:

```
pureview -batch -generate mti essram_denali -genoption  
init_file="../loadfiles/essram.dat" -genoutput essram.v essram.soma
```

This command would create a Verilog file called `essram.v` with a module name of “`essram_denali`” that references an initialization file of `../loadfiles/essram.dat` to preload the device and the `essram.soma` SOMA file. This HDL shell file (`essram.v`) will be generated for the MTI Verilog simulator.

Below is a complete list of the options available for the command line use of PureView.

TABLE 2-1: PureView Command line options

Option	Description
-batch	Open no windows, and exit after operations specified on the command-line.
-usage	Output usage information about PureView command
-help	Output help information about PureView command
-quiet	Do not write messages to the console
-new <class name>	Creates a new SOMA for the given memory class.
-convert <format>	Converts the input SOMA to the specified format. Valid options are "xml" and "uncompressedxml" (or "uxml").
-convoutput <file name>	Writes the converted SOMA (see -convert) to the given file instead of to the console.
-generate <target env> <model name>	Generate simulation source code for the given environment, using the given name for the Verilog module or VHDL entity. See list of target_environments below.
-genoption <option>[=<value>]	Set the named options to either the given values or to 1 where no value is supplied. The effects of multiple -genoption arguments on the command line are cumulative. Multiple options are comma separated, no spaces. See list of options below.
-genoutput <file name>	Write HDL source code (see -generate) to the given file instead of to the console.
-simdb <file name>	Connect to a simulation database file
-instance <instance name>	Instance name to view in the simulation database (requires -simdb)

TABLE 2-2: Target Environments for -generate option

Parameter	Environment
Co-Design	
CoWare	CoWare Simulator
Verilog	
all	All Verilog Simulators
VHDL	
configuration	VHDL Configuration Template
leapfrog	Cadence Leapfrog/NC-VHDL
mentor-unix	Mentor MTI Modelsim (UNIX)
mti-unix	MTI V-System/Mentor QuickHDL (UNIX)
package	VHDL Package Template
scirocco	Synopsys Scirocco
voyager	Ikos Voyager
Datasheet	
datasheet	HTML Datasheet
SystemC	
systemC	SystemC

TABLE 2-3: Parameters for -genoption

Parameter	Description
init_file"file name"	The value for the init_file parameter/generic. NOTE, you can use relative paths (../this_initfile.txt) as well as environment variables (\$denali_loadfiles/mem1.dat) in the file name.
bitblast[=<value>]	If set to any of 1, yes, or true, generate an additional bit-blasted HDL shell. NOTE: All busses will be blasted, for selective blasting, use the PureView GUI.
blastModelName= <model name>	Name of the module/entity for the bit-blasted or additional HDL shell (e.g. blastModelName=rbt12_75). Note that blastModelName is case sensitive.
combinesetuphold	If set to 1, it combines \$setup/\$hold conditions into one \$setuphold condition
configurationName	Name of the configuration for the VHDL configuration template
packageName	Name of the package for the VHDL package template
libraryName	Name of the library for the VHDL configuration template
vhdl_std_ulogic	If set to 1, VHDL port types can be either std_logic or std_ulogic (default is always std_logic).

3 Debugging Memory Using PureView

The PureView debugger provides a unique graphical view of your memory contents during simulation. It is intended for interactive as well as post-processing debugging and provides the following beneficial features for verification debug:

- Display physical and “logical addressing” contents
- Postprocessing database
- Interactively browse memory contents at any time after simulation has run
- Step forward or back in time while viewing memory contents
- Address-specific history
- Double-clicking on any memory location brings up transaction history for it
- Time-synchronization with simulator
- When running in “live” mode, can have PureView sync'ed with simulator
- User-selectable layout and display of memory space/contents

The PureView debugger is synchronized to your simulation using Denali’s client application interface. This ensures that the PureView window accurately reflects the data being read and written to memory

3.1 Denali Memory Database Files

To enable the post processing capability, you must specify a database file to be created. This file uses Denali’s advanced database compression technology to minimize performance impact during simulation. All memory events during the simulation are captured in this binary file.

This file can be specified at the beginning of simulation or at any time during simulation.

NOTE: *If the database file is specified after simulation has started, it may inhibit valuable information about system memory definition which might have preceded the simulation database creation.*

There are two methods for specifying the database file to be created.

1. At the beginning of simulation - Specify via *.denalirc* file
 - SimulationDatabase simdb
2. Dynamically during simulation - Specify in RTL through Tcl Interface
 - Verilog/VHDL - *mmtclevel("mmsimulationdatabase simdb_filename");*
 - Tcl - *mmsimulationdatabase simdb_filename*

It should be advised that using the *.denalirc* option is the preferred method, as it will capture the entire database from the beginning of simulation. If both methods are used, the *.denalirc* file will stop being recorded when *mmsimulationdatabase* is invoked.

3.2 Interactive Debugging During Simulation

As mentioned above, PureView can now be used in two modes: interactively during simulation and in post-processing modes. PureView can be invoked at any time before or during simulation. In either case, a database file must be generated. See Figure 3.1, “Denali Memory Database Files,” on page 30 for details on how to specify the database file.

3.2.1 UNIX Shell Invocation

To invoke PureView from the UNIX shell, use the following command:

```
unix> pureview &
```

Also, there are command line switches to bring up a specific database file and memory instance. These are used as follows:

-simdb filename - Will automatically connect PureView to a simulation database file.

-instance instance_name - Instance name to view in the simulation database.

Example: Open up instance “test.memory.ddr0” in the “test1.simdb” database.

```
pureview -simdb ./test1.simdb -instance test.memory.ddr0
```

The complete list of command line switches can be seen by typing:

```
pureview -batch -usage
```

Invoking PureView using “pureview &” will bring up the following client window:

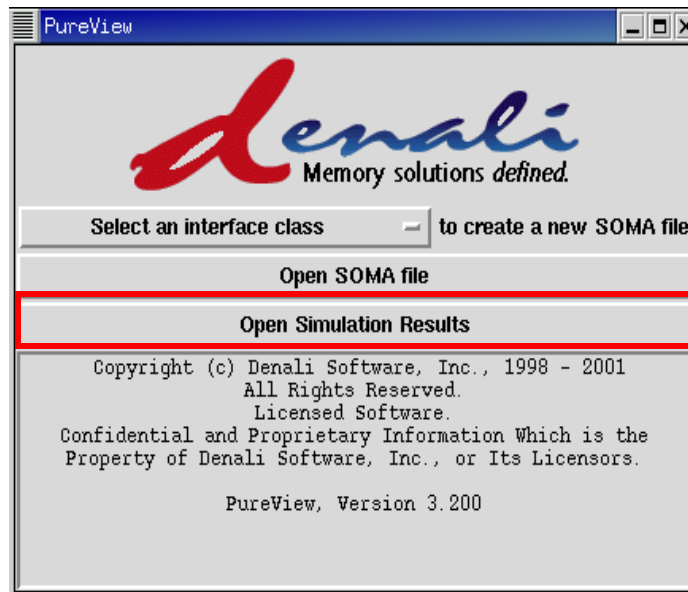


FIGURE 3-1: PureView Debugger Window

3.2.2 Simulator/Testbench Invocation

PureView can also be invoked directly from within your design/testbench. For convenience, Denali has provided PLI, FLI and TCL commands to invoke PureView directly from within your testbench or from your simulator’s command line.

For example:

```
ModelSim TCL: mmstartpureview
NC-SIM VHDL TCL : call mmtclevel "mmstartpureview"
Verilog PLI      : success = $mmstartpureview;
VHDL FLI (ModelSim only): success := mmstartpureview;
Verilog Command Line : $system("$DENALI/bin/pureview &");
```

3.3 Opening up the Simulation Results File

Once PureView is invoked, the database file must be opened to begin debugging with PureView. Click on the “**Open Simulation Results**” button to locate and open the simulation database file. Note that only ONE database file can be opened per PureView session.

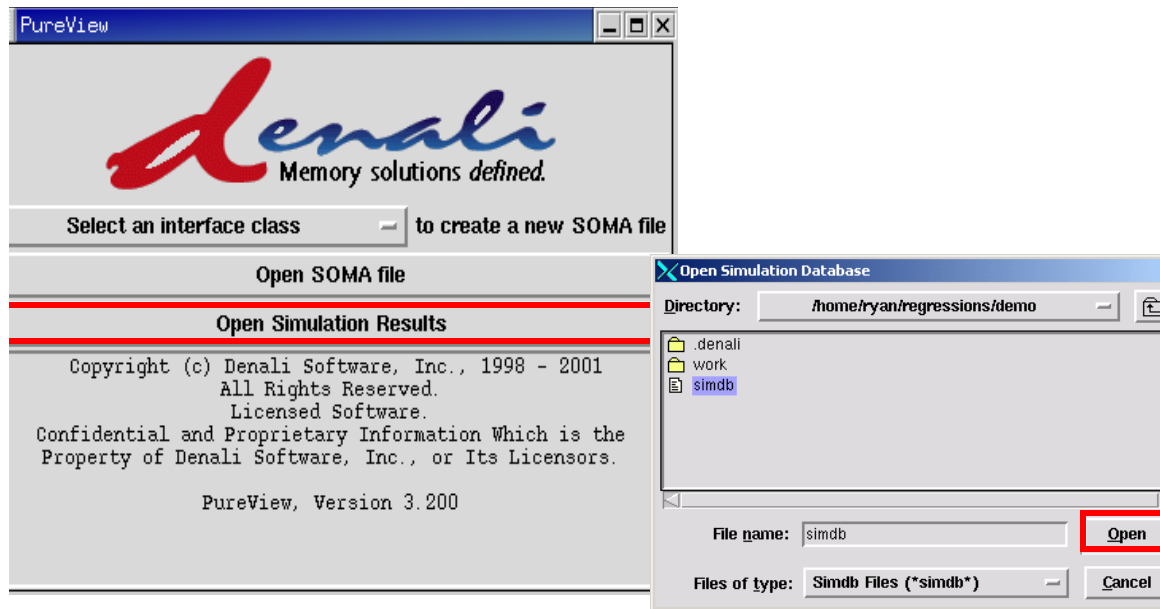


FIGURE 3-2: PureView “Open Simulation Results” Window

3.4 Selecting Memory Instances

PureView can be used to view either physical memory instances instantiated in your design or view “logically addressed” memories created using the features of MMAV. Logically addressed memories let you combine physical memories using depth, width, and interleave expansion to generate a more cohesive “system” view of your memory subsystem. Refer to the “Logical Address View” section in the Denali’s MMAV Guide for more information on creating these “logically addressed” memory views.

Once PureView has been invoked and the database file opened, a memory instance window is displayed. Select one (or multiple) memory instances and then click on “OK”.

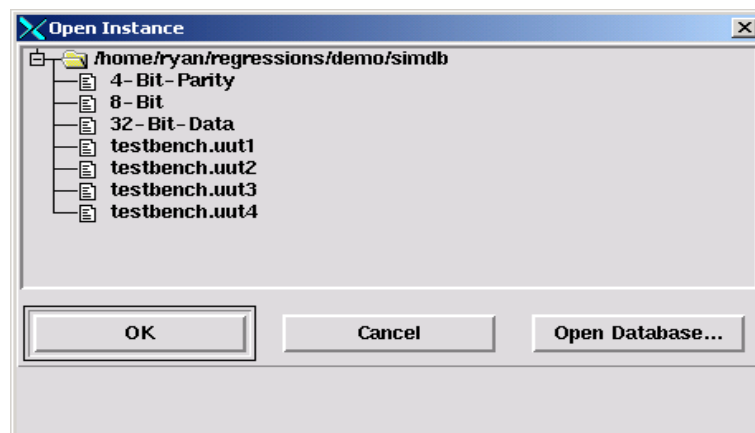


FIGURE 3-3: PureView “Open Instance” Window

3.5 PureView Debugging Windows

Once you have selected a memory instance to debug, two windows will be displayed. The first window, the Memory Contents Window, will display the contents of memory at the specified simulation time. The second window, the Transaction History Window, will display the sequential memory transactions that have been captured.

These two windows are synchronized with each other. For example, if you “Set” the time in the Memory Contents Window, the Transaction History Window will automatically be updated to show you the transactions at or near that point in time. Also, if you double click on a transaction in the Transaction History Window, the Memory Contents Window will automatically adjust to display the memory contents at that time in simulation.

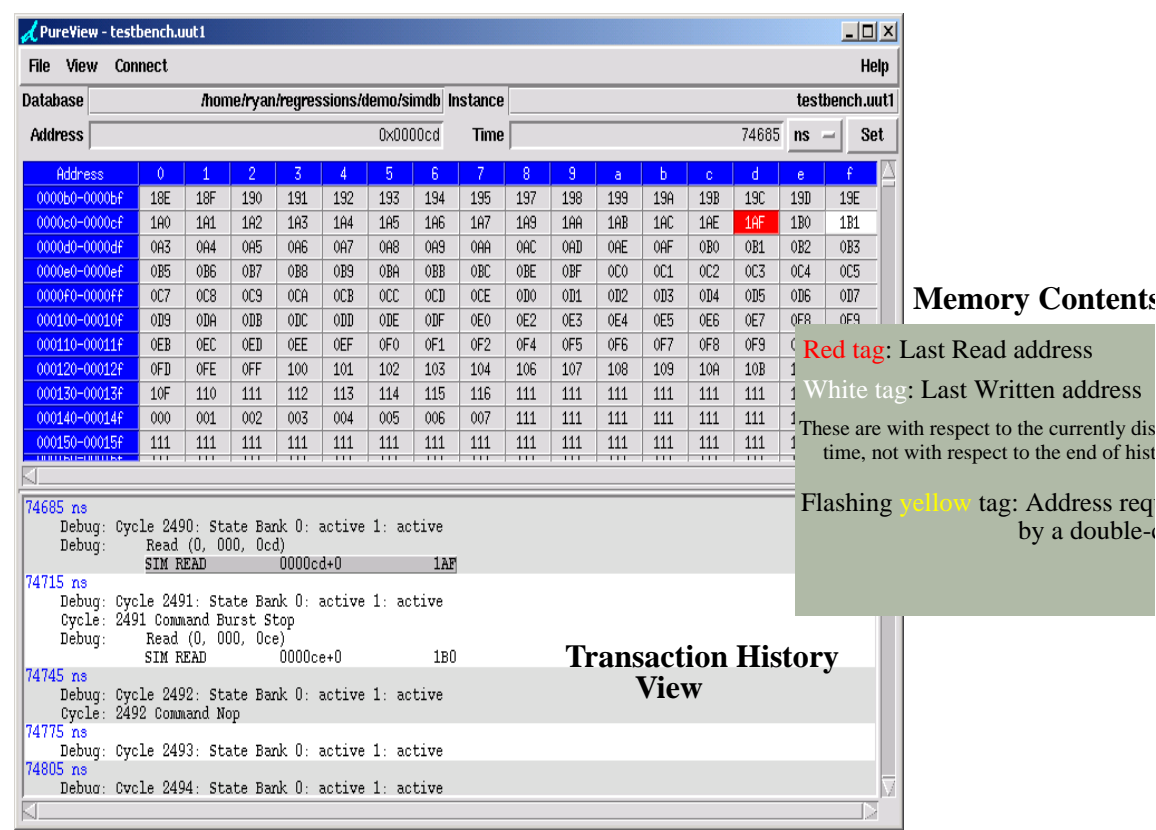


FIGURE 3-4: PureView Debugger Windows

3.5.1 Memory Contents Window

The Memory Contents Window has some additional features to help with debugging. These are activated by the pull-down menus. A close-up of this window is shown below.

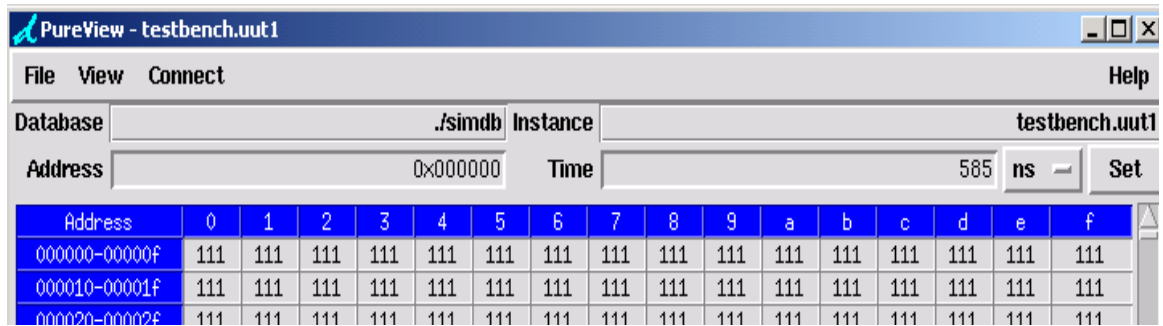


FIGURE 3-5: PureView Memory Contents Window Options

These features are described below.

From the **File** options pull-down menu, you can use the following features:

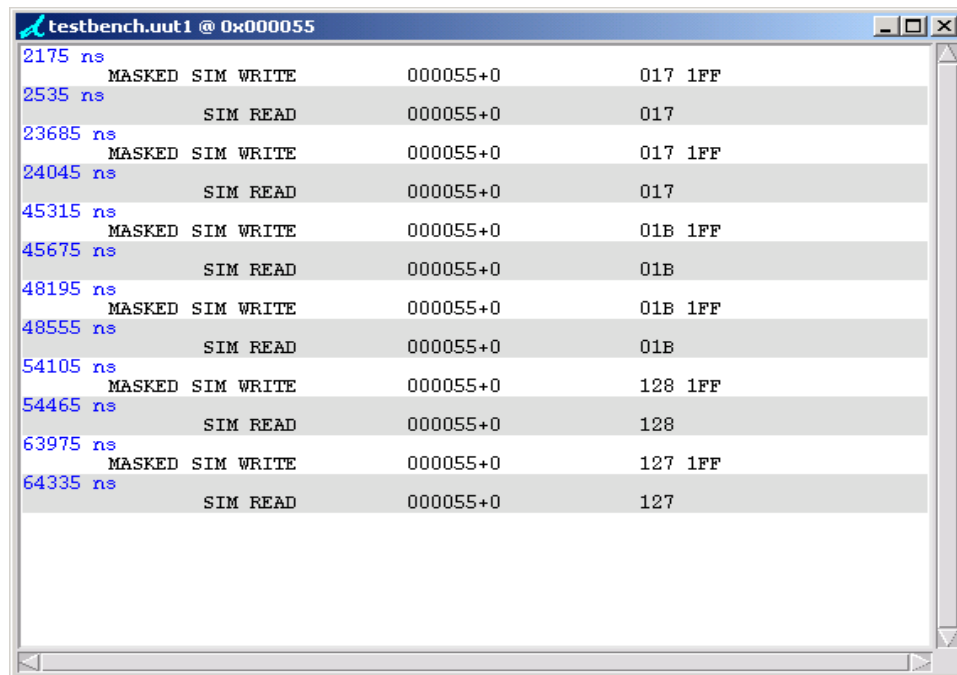
- **Open Instance...:** Opens the PureView memory instance window to allow the user to open additional debugger windows
- **Run Tcl Command...:** Allows user to issue a Tcl command, if desired.
- **Debug:** Turn on window debugging options to report a problem with the PureView GUI to Denali support.
- **Close Window:** Closes the current Memory Content Window
- **Exit:** Close all windows and exit PureView

From the **View** pull-down menu, you can use the following feature:

- **Value Format:** Allows you to set the format for the data values (addresses are always displayed in Hexadecimal format). You can select one of the following: Binary, Decimal, Hexadecimal, or Octal.
- **Contents Table Origin:**
 - **Top Left:** Display the memory contents window with the starting address starting at the top left
 - **Top Right:** Display the memory contents window with the starting address starting at the top right
 - **Bottom Left:** Display the memory contents window with the starting address starting at the bottom left
 - **Bottom Right:** Display the memory contents window with the starting address starting at the bottom right
- **Show Last Read:** Displays last Read address
- **Show Last Write:** Displays last Written Address
- **Refresh:** Refreshes the Memory Contents Window

3.5.2 Memory Contents Transaction Summary

You can get a summary of all the memory transactions for a specific address by “double-clicking” on that address. If there have been any memory transactions at that address, you will get the following pop-up window summarizing all the activity.



The screenshot shows a window titled "testbench.uut1 @ 0x000055". It contains a table of memory transactions. The table has four columns: time in nanoseconds (ns), operation type, address, and data. The transactions are as follows:

Time (ns)	Operation	Address	Data
2175	MASKED SIM WRITE	000055+0	017 1FF
2535	SIM READ	000055+0	017
23685	MASKED SIM WRITE	000055+0	017 1FF
24045	SIM READ	000055+0	017
45315	MASKED SIM WRITE	000055+0	01B 1FF
45675	SIM READ	000055+0	01B
48195	MASKED SIM WRITE	000055+0	01B 1FF
48555	SIM READ	000055+0	01B
54105	MASKED SIM WRITE	000055+0	128 1FF
54465	SIM READ	000055+0	128
63975	MASKED SIM WRITE	000055+0	127 1FF
64335	SIM READ	000055+0	127

FIGURE 3-6: PureView Transaction Summary Window

3.5.3 Transaction History View

PureView provides a complete and informative transaction history view to allow the user to see a list of all the memory transactions since PureView was invoked. The level of information in the transaction history window can be controlled by parameters in the *.denalirc* file (see “Controlling Your Memory Simulation Models - The *.denalirc* File” on page 39 for more details on the *.denalirc* file) as follows:

1. HistoryFile: Provides details on basic read/write operations and memory operations (precharge, nop, refresh, etc.)

2. HistoryDebug: In addition to the HistoryFile information, this option also provides detailed debug information such as bank/row/column addresses, bank states, etc.

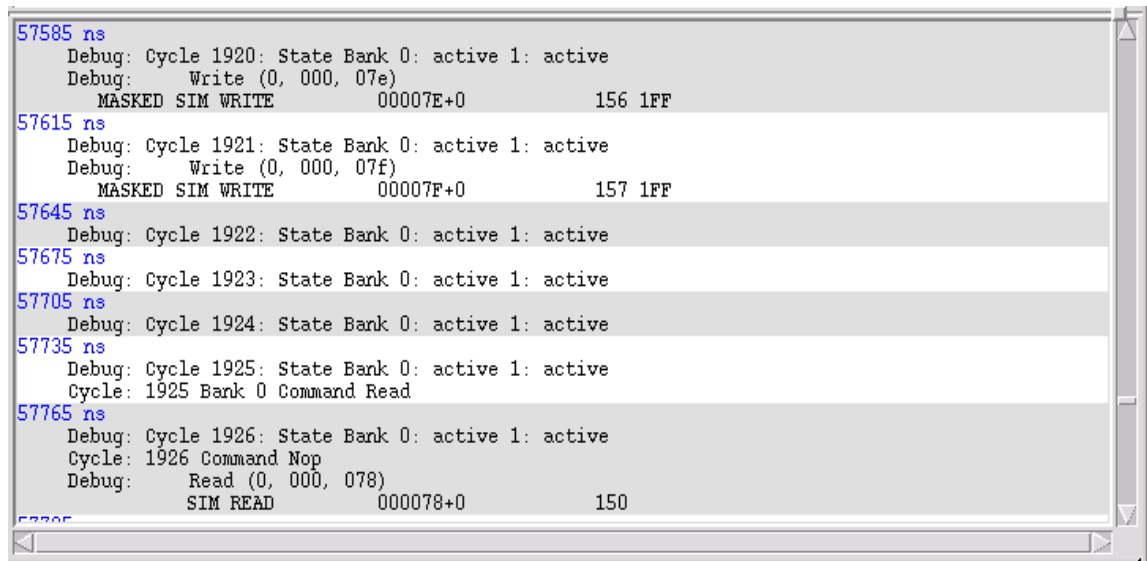


FIGURE 3-7: PureView Transaction History Window

Double-clicking on a line in the Transaction History window takes you to that time and address in the Memory Contents window.

3.6 Post-Processing with PureView

As mentioned above, PureView can now be used in two modes: interactively during simulation and in post-processing modes. To use PureView in post-processing mode, a database file must already exist. To use PureView in post processing mode, follow these steps:

1. Run your simulation with “SimulationDatabase” set either in the *.denalirc* file or from within your testbench
2. Invoke PureView and open the simulation database file
3. Open memory instances you wish to debug

All of the features mentioned above are still valid.

3.7 Using PureView with Mentor Graphic’s Seamless HW/SW Co-Verification

Prior to Denali’s 3.0 release, Seamless customers were unable to use Denali’s PureView debugger with Seamless. This was the result of Seamless “owning” the memories in the design. With Release 3.0, Denali and Mentor Graphics have solved this. There are two new *.denalirc* features called: **DenaliOwn** and **DenaliOwnClass** that pass control of the

memory within Seamless to Denali. This allows you to now use PureView to debug the memory contents.

NOTE: *To use PureView within the Seamless environment you must use these Denali settings to pass the ownership of the memories to Denali.*

Use the **DenaliOwn .denalirc** setting to pass a specific instance to Denali to “own”.

For example, to view the memory instance “testbench.rams.sdram0” in Seamless, set the **.denalirc** variable as follows:

```
DenaliOwn /testbench/rams/sdram0
```

You can also instruct Seamless to allow Denali to own an entire memory class. A memory class is the specific memory type (for example, sdram, ddr_sdram, flash, sram, etc.). For example, to view the sdram memory instances in PureView when running in Seamless, set the **.denalirc** variable as follows:

```
DenaliOwnClass sdram
```

NOTE: *In order to add multiple memory classes to the **DenaliOwn** and **DenaliOwnClass** parameter, you **MUST** separate the memory classes by using a semicolon and ensure that these are on the same line of text.*

Examples:

```
DenaliOwn /testbench/rams/sdram0;/testbench/rams/sdram1;*ddr*  
DenaliOwnClass sdram;ddr
```

4 Using Denali's Memory Modeler Advanced Verification

Denali's Memory Modeler Advanced Verification (MMAV) product extends Denali's world class memory modeling capability with advanced verification features aimed at enhancing your memory subsystem verification. Using the data and data transactions to verify your system can significantly reduce your verification cycle time while increasing your verification coverage.

4.1 Controlling Your Memory Simulation Models - The .denalirc File

In order to achieve the most out of MMAV, Denali uses a runtime control initialization file, *\$DENALI/.denalirc*, which stores settings to control model behavior during simulation. Up to four *\$DENALI/.denalirc* files may be used to store these settings. For example, if you only wish to change one setting for a particular simulation, you would create a *.denalirc* file in your working directory to store your specific simulation settings. The locations of these initialization files are listed below in order of precedence:

TABLE 4-1: .denalirc Precedence

Location	Description
<i>\$DENALIRC</i>	Environment variable
<i>./denalirc</i>	Simulation specific defaults
<i>~/denalirc</i>	User defaults
<i>\$DENALI/.denalirc</i>	System defaults

In general, the *.denalirc* file is a simple text file containing keyword-value pairs used to store your initialization settings. All lines beginning with *#* are ignored. The default *.denalirc* file in your installation also includes commented lines which provide a complete description of all the switches and their usage. The descriptions use mixed case for clarity but the switch names are NOT case sensitive, though their values may be.

The following section describes some of the switches (or flags) that can be set in the initialization file to modify the behavior of your memory models.

HistoryFile

When set, the History File switch creates a history file containing all read and write operations to each memory. In your default *.denalirc*, this switch is turned off. To set this switch, simply un-comment the appropriate line, and specify a file to which the history should be written to. For example, to save your history file as “memresults.his”, the line in your *.denalirc* file would read as follows:

```
HistoryFile memresults.his
```

HistoryDebug

The HistoryDebug switch causes more detailed information to be saved to your History-File results. In your default *.denalirc*, this switch is turned off. To set this switch, simply un-comment the appropriate line, for example:

```
HistoryDebug On
```

HistoryDebugLoad

The HistoryDebugLoad switch causes even more information to be saved to your History-File results, including each individual address load to the file. Consequently, setting this switch will greatly increase the size of your HistoryFile. In your default *.denalirc*, this switch is turned off. To set this switch, simply un-comment the appropriate line, for example:

```
HistoryDebugLoad On
```

HistoryInSimLog

Use HistoryInSimLog set to 1 to see the history messages reflected in wherever the output from the simulator is going (which varies by simulator). This interleaves Denali history with testbench output. The default for this is 0 (off).

```
HistoryInSimLog 0
```

TraceFile

When set, the TraceFile switch creates a file containing all events on the memory ports such as reads, writes, etc. This is primarily used by Denali support as a valuable diagnostic tool for understanding and recreating a customer’s memory simulation environment. In your default *.denalirc*, this switch is turned off. To set this switch, simply un-comment the appropriate line, and specify a file to which the trace file information should be written to. For example, to save your trace file as “testcase.trc”, the line in your *.denalirc* file would read as follows:

```
TraceFile testcase.trc
```

TraceTimingChecks

TraceTimingChecks enables you to trace all of the timing checks performed during simulation. In your default *.denalirc*, this switch is turned off. To turn on TraceTimingChecks, simply set the value to 1, for example:

TraceTimingChecks 1

LicenseQueueTimeout

LicenseQueueTimeout allows you to specify how many minutes to wait suspended, if a license is not available. In the example below, Denali will wait for 120 minutes for a license to become available.

LicenseQueueTimeout 120

LicenseQueueRetryDelay

LicenseQueueRetryDelay specifies how many seconds to wait before pinging for licenses (so that the license log file doesn't overflow). In the example below, Denali will ping every 60 seconds for a Denali license.

LicenseQueueRetryDelay 60

SimulationDatabase

SimulationDatabase specifies the filename and location for the post-processing simulation database.

SimulationDatabase /tmp/simdbxxx

Where XXX is a unique number to avoid collisions with other users. You can specify a different database location than above if desired. However, for performance reasons it is strongly recommended to locate the database on a local disk.

NOTE: *You must turn on **HistoryFile** parameter when generating a database.*

To turn off database generation (this is the default), use:

SimulationDatabase Off

SimulationDatabasePattern

To limit the database info to specified instances, use the statement:

SimulationDatabasePattern instance2

SimulationDatabaseBuffering

You can turn the simulation database buffering on/off if you want database flushed to disk immediately. This results in a performance hit but is useful if you're debugging an "abnormal termination" situation:

SimulationDatabaseBuffering on/off

TimingChecks

The TimingChecks switch turns on/off the timing checks on your memory model (i.e. setup times, hold times, etc.) In your default *.denalirc*, this switch is turned on. To turn off timing checks, simply set the value to 0, for example:

```
TimingChecks 0
```

RefreshChecks

The RefreshChecks switch turns on/off the refresh timing checks for DRAM models. If you are not simulating with DRAM models, this switch is ignored. In your default *.denalirc*, this switch is turned on. To turn off refresh checks, simply set the value to 0, for example:

```
RefreshChecks 0
```

RefreshOnReadWrite

If you would like to count reads and writes as a refresh to that particular row, set the following parameter RefreshOnReadWrite to 1. Although the true behavior of the part may count a read and write as a refresh to that row, Denali does not default to that behavior. The reason being that a controller should not rely on certain memory accesses to obtain proper refresh rates. This is *only* supported for edram, rldram, sdram, ddr sdram, and ddr II sdram memories.

```
RefreshOnReadWrite 1
```

ReadDQSContentionCheck

To have the model check for bus contention on DQS during reads, set ReadDQSContentionCheck to '1'. Supported for DDR SDRAM only. DDR SDRAM model checks for bus contention on DQS during reads. To disable this check, set ReadDQSContentionCheck to '0'.

```
ReadDQSContentionCheck 1
```

InitialMemoryValue

By default, your memory models are initialized to X. You may specify any of the following values to initialize your memory:

- InitialMemoryValue 0 - initializes to all 0's
- InitialMemoryValue 1 - initializes to all 1's
- InitialMemoryValue X - initializes to all X's
- InitialMemoryValue U - initializes to all 'U's

Alternatively, you can also specify a hex value. This value must begin with "0x" and consist of the hex values 0-9 and A-F (or a-f). For example:

```
InitialMemoryValue 0x3018
```

This string cannot expand to be longer than the memory's word width (typically the data size). If shorter, the unspecified bits will be filled with 0s.

Any other value will initialize memory to all 'X's

You may also generate random data for the initial data by specifying:

- `InitialMemoryValue randomNoUpdate` - the model is not updated with the random data (i.e. it is not written to the model). So subsequent reads without intervening writes will get new random data.
- `InitialMemoryValue randomWithUpdate` - the model is updated with the random data, (i.e. it is written to the model). So subsequent reads without intervening writes will get the same random data that was returned the first time.

Alternatively, you can use any arbitrary C function to specify the initial data, including random data, random data with parity, etc.

For examples, refer to `$DENALI/ddvapi/example/fillValue`.

InitMessages

`InitMessages` causes the system to report an informative message concerning each memory component instantiated in your design during initialization. In your default `.denalirc`, `InitMessages` is turned on. To turn off `InitMessages`, simply set the value to ***Off***, for example:

```
InitMessages Off
```

TracePattern

`TracePattern` allows you to limit the size of your trace file (see above) by limiting the capture to specific instance name parameters. You may use shell “glob” patterns such as `*`, `?`, `[]`. You **MUST** have `TraceFile` uncommented as well.

For example, to trace just memory instances with the pattern “sdram”, you would use:

```
TracePattern *sdram*
```

HistoryPattern

`HistoryPattern` allows you to limit the size of your history file (see above) by limiting the capture to specific instance name parameters. You may use shell “glob” patterns such as `*`, `?`, `[]`.

For example, to record history for just memory instances with the pattern “sdram”, you would use:

```
HistoryPattern *sdram*
```

IrregularClock

In models with random output delay scheduling, such as DDR SDRAM, DDR-II, FCRAM, RDRAM, and RLDRAM, output scheduling and some timing checks are affected by the actual clock cycle time. The clock cycle time is typically measured by the model during the first few cycles of simulation only, unless one of the following flags is used.

This feature must be set if you are running with uneven clocks (non-constant clock-widths). If you are running with regular (even) clocks, the Denali models can “randomize” the data output within the allowable data valid range (see **RandomOutputDelay** below). If **IrregularClock** is set to ‘1’, then this randomization will be automatically turned off.

When this is set, the model measures the clock width every cycle and disables random output delay scheduling.

```
IrregularClock 1
```

ClockStableCycles

This is used to define the number of cycles in a row where MMAV considers clock stabilized. This is necessary for DDR, FCRAM and RLDRAM as there is typically a period of time before the clocks become stabilized.

```
ClockStableCycles 1000
```

RandomOutputDelay

This parameter is used in models with random output delay scheduling (DDR, FCRAM, DDR, etc.), which better exercises the memory controller. You might want to turn it off early in your verification cycle. By default this value is on.

```
RandomOutputDelay 1
```

OutputTiming

By default, Denali memory models drive the data outputs with delay based on SOMA file parameters. If you wish to drive the model outputs with zero-delay, the **OutputTiming** *.denalirc* variable must be set to “0”. This is primarily used in cycle based simulations.

```
OutputTiming 0
```

InitChecks

DRAM initialization checks can be disabled using the **InitChecks** parameter. If the initialization checks are turned off, Denali will NOT check for the proper DRAM initialization sequence. Use this command with caution as real errors may be masked if this is turned off.

To turn off the initialization checks.

```
InitChecks 0
```

InitChecksPauseTime

This variable is used by the Denali model to ignore tpause/tinit checks in DRAM models, and DLL Lock time checks in SSRAM models.

```
InitChecksPauseTime 0
```

ErrorMessages

This *.denalirc* option can be used to turn on/off warning and error messages globally. Errors and warnings will still be captured in Denali history and trace files, but messages will be purged from the simulator output/console.

```
ErrorMessages on  
ErrorMessages off
```

ErrorMessagesStartTime

Denali also provides the capability to turn on error and warning messages at a specific time. This allows time for reset and device initialization before Denali models start reporting errors. Errors and warnings will still be captured in Denali history and trace files, but messages will be purged from the simulator output/console. See examples below for syntax.

```
ErrorMessagesStartTime 0ps      # default  
ErrorMessagesStartTime "20 us"  
ErrorMessagesStartTime "200ms"  
ErrorMessagesStartTime 200000 ns
```

ExitOnErrorCount

This *.denalirc* variable will allow you to exit simulation when a specified number of Denali errors have occurred. Simply specify the error threshold in the *.denalirc* file:

```
ExitOnErrorCount 10  #exits simulation after 10 Denali errors
```

ErrorCount

In Verilog code, you can use an integer variable to see the number of errors detected by the Denali memory models. Declare an integer variable in your testbench and then register it as the error count variable through the ErrorCount switch in the *.denalirc* file. Then our models will increment this variable each time an error detected. You may monitor the error count variable, branch on it, etc.

Sample usage:

In testbench:

```
module testbench;  
integer errCount;      // monitor, display this
```

In *.denalirc*: uncomment the following line:

```
ErrorCount testbench.errCount
```

TclInterp

This setting is used by NC Sim users. By default, Denali's Tcl interpreter with NC Sim is turned off due to some earlier incompatibilities with NC Sim's Tcl interpreter. To enable Denali's Tcl interpreter in NC Sim, you must explicitly enable it by this *.denalirc* setting.

```
TclInterp 1
```

TrackAccessFromInit

If you are preloading memory with "init_file" or `mmload` before setting breakpoints and you want these actions to be counted as memory write accesses, then set the following variable: (default is 0)

```
TrackAccessFromInit 1
```

This is useful when setting breakpoints on unaccessed memory locations.

EiMessages

Suppresses messages for soft error injections. By default, all injected errors are reported.

```
EiMessages off
```

DifferentialClockChecks

This setting suppresses differential clock checking. By default, models check negative polarity clock signals for synchronization with positive polarity clock signals.

```
DifferentialClockChecks 0
```

DifferentialClockSkew

This setting allows you to specify allowable skew between positive and negative polarity clock signals for differential clock checking. Skew is measured from the time either one of the signals switches to the time the opposite signal switches. By default, no skew is allowed. Note that in an actual device, differential clock skew is meaningless since the clock edge is defined as the crossing of the positive and negative clock signals. This parameter accounts for the fact that rise and fall times are not modeled in simulation, and its value is not provided by vendors. Skew is specified using a time value and units.

```
DifferentialClockSkew 150 ps
```

AssertionMessages

Suppress the messages displayed when an assertion fires. By default, a message is printed when an assertion is triggered.

AssertionMessages off

TraceBackdoorReadWrite

Suppress the trace for backdoor reads/writes. By default it is traced in the Denali trace file (if specified).

TraceBackdoorReadWrite 0

DenaliByPass

Renders all Denali models as non-functional, and does not check out a license during simulation. The default value is 0.

DenaliByPass 1

4.1.1 Register File Specific .denalirc Parameters

SuppressUnknownAddrReadError

Suppresses the error messages when an unknown address is read from the instance whose name matches "InstNamePattern".

SuppressUnknownAddrReadError InstNamePattern

4.1.2 IBM-EDRAM Specific .denalirc Parameters

SuppressRefreshInfoMessages

Use 'SuppressRefreshInfoMessages 1' to eliminate informational messages about the actual refresh window size (when it's not an error or warning). The default is '0', or to report on every refresh cycle.

SuppressRefreshInfoMessages 0

4.1.3 RDRAM Specific .denalirc Parameters

WarnSuppress

In addition to normal timing and protocol error conditions, the Denali RDRAM model issues warnings for the following hazardous operations:

- overwriting the write buffer before it's retired (e.g. WR-WR-RD-RD-RTR).
- precharging a bank before it's retired (rule CR8).

These warnings can be suppressed by setting the WarnSuppress parameter in the *.denalirc* file as follows:

WarnSuppress 1

TimingChecksReportOnly

Turning this *.denalirc* option on will disable the Denali model feature that corrupts memory and drives “X’s” on the data-bus when there are timing errors. This option allows timing errors to only cause the model to issue messages and will NOT drive “X” when seeing timing errors. This can be very useful in early evaluation of errors, but will allow the simulation to continue, though reporting errors.

TimingChecksReportOnly 1

TimingChecksStartTime

Denali also provides the capability to turn on timing checks at a specific time. This allows time for reset and device initialization before Denali models start checking for timing errors. See examples below for syntax.

```
TimingChecksStartTime 0ps
TimingChecksStartTime "20 us"
TimingChecksStartTime "200ms"
TimingChecksStartTime 200000 ns
```

4.1.4 RLDRAM Specific .denalirc Parameters

RldramInitCyclesCheck

To turn off the 2000 cycle check between each refresh during initialization.

RldramInitCyclesCheck 0

RldramInitRefreshChecks

To turn off refresh checking to each bank at initialization, set this to “0”.

RldramInitRefreshChecks 0

InitMrsAddressStable

This setting is used to enable address stability checks during Init MRS. It can be forced to check it by setting "InitMrsAddressStable 1" as follows:

InitMrsAddressStable 1

4.1.5 DDR-II SDRAM Specific .denalirc Parameters

OffChipDriveImpedanceChecks

The OffChipDriveImpedanceChecks variable is used to disable OCD checking. When this variable is 0, the model does not issue warning messages or corrupt data when the OCD level is outside the valid range. Note that there's also a SOMA feature that enables OCD. If the SOMA feature is disabled, OCD is completely ignored by the model.

```
OffChipDriverImpedanceChecks 0
```

MRSmsgsInSimLog

MRS/EMRS informative messages are always included in the history and the simulation log/transcript file. If you do not want to see these messages in the simulation log, then set the MRSmsgsInSimLog parameter to 0. By default, this parameter is set to “1”.

```
MRSmsgsInSimLog 0
```

4.1.6 DDR-II and DDR3 Specific .denalirc Parameters

noXsInReadData

Invalid ranges of model driven read data are not padded X's.

```
noXsInReadData 0
```

RandomInsteadOfXInReadData

Invalid ranges of model driven read data padded with rand data not X.

```
RandomInsteadOfXInReadData 0
```

4.1.7 ESSRAM Specific .denalirc Parameters

SuppressPortContention

The SuppressPortContention variable is used to suppresses Read-Write Port contention error messages in instances whose name matches the pattern specified by "InstNamePattern".

```
SuppressPortContention InstNamePattern
```

4.1.8 Mentor Graphics ModelSim Specific .denalirc Parameters

ModelSimTimeDefinitionToggle

Denali tries to be intelligent about interpreting high and low bits of time for the ModelSim simulator. This behavior is platform and release dependent in ModelSim. Denali's automatic behavior may not be appropriate in all cases, so the following switches give the user

control over the automatic behavior of Denali. To override the automatic interpretation of high and low time bits and turn off the toggling behavior use:

```
ModelSimTimeDefinitionToggle off
```

To make toggling the default behavior (avoids certain messages), use:

```
ModelSimTimeDefinitionToggle on
```

4.1.9 Mentor Graphics Seamless HW/SW Co-Verification Specific .denalirc Parameters

Prior to Denali's 3.0 release, Seamless customers were unable to use Denali's PureView debugger with Seamless. This was the result of Seamless "owning" the memories in the design. With Release 3.0, Denali and Mentor Graphics have solved this. There are two new *.denalirc* features called: DenaliOwn and DenaliOwnClass that pass control of the memory within Seamless to Denali. This allows you to now use PureView to debug the memory contents.

To use PureView within the Seamless environment you must use these Denali settings to pass the ownership of the memories to Denali.

DenaliOwn

Use the DenaliOwn *.denalirc* setting to pass a specific instance to Denali to "own".

For example, to view the memory instance "testbench.rams.sdram0" in Seamless, set the *.denalirc* variable as follows:

```
DenaliOwn /testbench/rams/sdram0
```

DenaliOwnClass

You can also instruct Seamless to allow Denali to own an entire memory class. A memory class is the specific memory type (for example, SDRAM, DDR_SDRAM, Flash, SRAM, etc.). For example, to view the SDRAM memory instances in PureView when running in Seamless, set the *.denalirc* variable as follows:

```
DenaliOwnClass sdram
```

4.1.10 OneNand Flash .denalirc Parameters

minimizeCallbacks

The oneNand Flash has a buffer-RAM and a Flash array when the data is moved from the buffer-RAM to the Flash array or from the Flash array to the buffer-RAM. If minimize-Callbacks is enabled then only callbacks that are applicable will occur. For example, if

you issue a program command to read the buffer-RAM and write to the Flash array, and if minimize callbacks is enabled, then only the write to the Flash array callbacks will occur.

```
minimizeCallback 1
```

oneNandEnableReadArrayCBs

If this *.denalirc* variable is enabled, and for example a load is performed, a true read is done from a Flash array. This will cause a callback to be issued for each read of the Flash array.

```
oneNandEnableReadArrayCBs 1
```

4.1.11 .denalirc Summary Table

TABLE 4-2: .denalirc Keywords

Keyword	Values	Default	Description
HistoryFile	filename	None	Saves a file with Denali read/write information
HistoryDebug	On/Off	Off	Adds additional debug information to the file created using "HistoryFile"
HistoryDebugLoad	On/Off	Off	Adds additional information regarding each address if it loaded from a file
HistoryInSimLog	0/1	0	Redirects the history file to a user's simulation log vs. the filename mentioned in "HistoryFile"
TraceFile	filename	None	Saves "trace" information to a file. Used primarily by Denali support to debug customer problems
TraceTimingChecks	0/1	0	Adds additional timing check information to the file created using "TraceFile"
LicenseQueueTimeout	time value (in minutes)	None	LicenseQueueTimeout allows you to specify how many minutes to wait suspended, if a license is not available
LicenseQueueRetryDelay	time value (in seconds)	None	LicenseQueueRetryDelay specifies how many seconds to wait before pinging for licenses (so that the license log file doesn't overflow).
SimulationDatabase	filename	Off	Setting this parameters defines the file for the simulation database. NOTE- you MUST also have HistoryFile set as well
SimulationDatabasePattern	pattern string	All instances	Adds only information about the specified "pattern" to the simulation database file. Used to limit the amount of information dumped into the database file.
SimulationDatabaseBuffering	On/Off	Off	Turn simdb buffering off if you want database flushed to disk immediately. This results in a performance hit but is useful if you're debugging an "abnormal termination" situation.
TimingChecks	0/1	1	Check for setup/hold timing violations
RefreshChecks	0/1	1	Check for DRAM refresh violations
RefreshOnReadWrite	0,1	0	If you would like to count reads and writes as a refresh to that particular row, set the following parameter RefreshOn-ReadWrite to 1. Although the true behavior of the part may count a read and write as a refresh to that row, Denali doesn't default to that behavior. The reason being that a controller should not rely on certain memory accesses to obtain proper refresh rates. This is <i>only</i> supported for edram, rldram, sdram, ddr sdram, and ddr ll sdram memories.
ReadDQSContentionCheck	0,1	0	Instruct the model to check for bus contention on DQS during reads
InitialMemoryValue	X,1,0	X	The initial value for all the memories
InitMessages	Off/On	On	Turn on/off messages pertaining to the instances in the design
TracePattern	pattern string	All instances	Adds only information about the specified "pattern" to the trace file. Used to limit the amount of information dumped into the trace file.
HistoryPattern	pattern string	All instances	Adds only information about the specified "pattern" to the history file. Used to limit the amount of information dumped into the history file.

TABLE 4-2: .denalirc Keywords

Keyword	Values	Default	Description
IrregularClock	0/1	0	If your clock is non-periodic, this must be set to 1, resulting in data aligned with clock edges.
ClockStableCycles	number of cycles	0	Specifies the number of stable clock cycles used to determine the clock cycle time
RandomOutputDelay	0/1	1	If IrregularClock is '0', then this parameter will enable/disable the output randomization for certain memory devices
OutputTiming	0/1	1	By default, the memory model drives output with delay, if set to 0, then it drives output with zero delay, mainly used for cycle-based simulations
InitChecks	0/1	1	Turn off/on initialization checks.
InitChecksPauseTime	0/1	1	This variable is used by the Denali model to ignore tpause/tinit checks in DRAM models, and DLL Lock time checks in SSRAM models.
ErrorMessages	On/Off	On	Is used to turn off error messages completely. Messages will still be logged in the History and Trace files if these are enabled.
ErrorMessagesStartTime	time value	0 ns	Is used to specify a start time for reporting error messages. Useful to avoid messages during initialization and reset. Messages will still be logged in the History and Trace files if these are enabled.
ExitOnErrorCount	variable name	None	Verilog feature that allows a user to set a variable to query within the HDL code to check the number of errors that have occurred.
ErrorCount	variable name	None	Verilog feature that allows a user to set a variable to query within the HDL code to check the number of errors that have occurred.
TclInterp	0/1	0	Enables/Disable Denali's Tcl interpreter when using NC-Sim
TrackAccessFromInit	Off/On	Off	Allows preload and backdoor operations to count as memory "write" accesses for breakpoints on uninitialized memory accesses
EiMessages	On/Off	On	Disables error messages for error injections
DifferentialClockChecks	0/1	1	Suppresses differential clock checking. By default, models check negative polarity clock signals for synchronization with positive polarity clock signals.
DifferentialClockSkew	time value and unit	0 ns	Allows you to specify allowable clock skew between positive and negative polarity clock signals for differential clock checking.
AssertionMessages	on/off	on	Model displays or suppresses assertion messages when a Denali assertion is triggered. Default is a message.
TraceBackdoorReadWrite	0/1	1	Allows user to suppress backdoor read/write messages from the trace file. By default, the messages are generated in the tracefile.
Register File Specific Parameters			
SupressUnknownAddrReadError	instance pattern string	No instances	Suppresses the error messages when an unknown address is read from the instance whose name matches the "instance pattern string"
IBM eDRAM Specific Parameters			
SuppressRefreshInfoMessages	0/1	0	Eliminate informational messages about the actual refresh window size (when it's not an error or warning). The default is '0', or to report on every refresh cycle.

TABLE 4-2: .denalirc Keywords

Keyword	Values	Default	Description
RDRAM Specific Parameters			
WarnSuppress	0/1	0	Suppresses warning messages for specific protocol checks (see details above)
TimingChecksReportOnly	0/1	0	Disables the Denali model feature that corrupts memory and drives "X's" on the data-bus when there are timing errors
TimingChecksStartTime	time value	0 ps	Is used to specify a start time for reporting error messages. Useful to avoid messages during initialization and reset. Messages will still be logged in the History and Trace files if these are enabled.
RLDRAM Specific Parameters			
RldramInitCyclesCheck	0/1	1	Checks for 200 cycles between refresh commands during initialization.
RldramInitRefreshCheck	0/1	1	Turn off refresh checking to each bank during initialization
InitMrsAddressStable	0/1	0	This setting is used to enable address stability checks during Init MRS.
DDR-II SDRAM Specific Parameters			
OffChipDriveImpedanceChecks	0/1	1	Disables OCD checking. When this variable is 0, the model does not issue warning messages or corrupt data when the OCD level is outside the valid range. Note that there's also a SOMA feature that enables OCD. If the SOMA feature is disabled, OCD is completely ignored by the model.
eSSRAM Specific Parameters			
SuppressPortContention	pattern string	none	Suppresses Read-Write Port contention error messages in instances whose name matches the pattern specified by "InstNamePattern".
ModelSim Simulator Specific Parameters			
ModelSimTimeDefinitionToggle	on/off	on	Automatic interpreting of the high and low bits of time for the ModelSim simulator
Denali PureView/Mentor Seamless Specific			
DenaliOwn	/path/to/memory/instance	none	Specifies a particular instance to Seamless that Denali "owns" for debug
DenaliOwnClass	ddr_sdram, sdram, flash, sram, and other memory classes	none	Specifies a particular memory class type that Denali "owns" for debug
SaveDataRadix	2, 8, 10, or 16	16	Specifies the format setting as 2 (binary), 8 (octal), 10 (decimal), or 16 (hexadecimal) for the mmsave* data.
SaveAddressRadix	2, 8, 10, or 16	16	Specifies the format setting as 2 (binary), 8 (octal), 10 (decimal), or 16 (hexadecimal) for the mmsave* address.

4.2 Setting .denalirc Options Dynamically During Simulation

In Release 3.0, Denali added a powerful feature to allow users to control *.denalirc* settings directly from their testbench. A Tcl command called “**mmsetvar**” was added. To use **mmsetvar** within your testbench, the Denali PLI/FLI call **mmtclevel** command can be

called to interpret the Tcl command **mmsetvar**. For example, to dynamically turn on history file generation (creating a history file named `denali.his`) from your testbench, you would use the following syntax:

Verilog:

```
success = $mmtcleval("mmsetvar historyfile denali.his");
```

VHDL:

```
success := mmtcleval("mmsetvar historyfile denali.his");
```

TCL:

```
mmsetvar historyfile denali.his
```

4.3 Licensing Solutions

4.3.1 Simulator Queuing for Denali Licenses

License queuing allows your simulation to “wait” for a Denali license if one is unavailable when your simulation is loaded. License queueing can be turned on by setting the environment variable: `DENALI_LICENSE_QUEUE_TIMEOUT` in your shell environment, and specifying the number of minutes to wait for a Denali license. For example to queue for up to 60 minutes, use:

```
setenv DENALI_LICENSE_QUEUE_TIMEOUT 60
```

By default, this feature is OFF.

To specify seconds to wait before pinging for MMAV license (so that the license log file does not overflow), in your environment set the following:

```
setenv DENALI_LICENSE_QUEUE_RETRY_DELAY 60
```

4.3.2 Speeding up License Checkout

Users can now specify Denali-related license files in an environment variable called `DENALID_LICENSE_FILE`. For example:

```
setenv DENALID_LICENSE_FILE /home/denali/licenses/license.dat
```

In this case Denali will only search the license server associated with the server hosting the “`/home/denali/licenses/license.dat` file”. In this case, the `LM_LICENSE_FILE` environment variable is ignored. This can significantly speed up license checkouts as only license servers that serve up Denali licenses are checked. Also, commas are now supported as separators in license file paths.

4.4 The Denali Tcl Interface

Denali provides a direct Tcl interface for all of its functions. Tcl is becoming more prevalent as a verification language and is very powerful. Tcl is also portable across simulators as VHDL simulators do not share a common C interface as Verilog simulators do, as most simulators support Tcl. All of the Denali functions to be discussed later in this document have been ported to Tcl for ease of use.

4.4.1 Using with ModelSim

Because ModelSim has a built-in Tcl interpreter, all Denali Tcl commands can be executed directly from the ModelSim command line or from a ModelSim **.do** file.

Example:

```
mmwriteword testbench.uut1 0 11110000
```

4.4.2 Using with NCSIM and other Tcl Interpreters

Because NCSIM (and other simulators) have their own Tcl interpreters built-in, you must load the Denali library **libmmcalls.so** from any Tcl interpreter to allow the two Tcl interpreters to communicate with each other. Once this is done, all the Denali **mm*** calls are available as Tcl commands as in ModelSim:

Example:

```
ncsim> load /<denali path>/libmmcalls.so
ncsim> set x [mminstanceid .....]
ncsim> set a [mmreadword .....]
```

Alternatively, you can use the Denali CFC library as another method for externally calling a limited number of Denali C foreign function routines such as **mmtclevel**. The **lib-dencfc.so** library is located in \$DENALI with a bootstrap function named **den_CFCPtr**. However, Denali recommends loading **libmmcalls.so** because the Denali Tcl calls get loaded within the simulator's Tcl interpreter. In this way, the function calls and variables will be directly visible to the simulator interpreter as opposed to using Denali's Tcl interpreter which is external.

4.4.3 Tcl Commands

mmtclevel - This command is used within a VHDL or Verilog testbench to evaluate another Denali Tcl command. This command will return a "0" value if successful and a "-1" value if an error was encountered.

Examples:

Use Tcl to do a “back-door” read from memory instance /testbench/top/sdram at address=0.

Verilog

```
success = $mmtcleval("set memid [mminstanceid testbench.top.sdram]");  
success = $mmtcleval("mmreadword $memid 0");
```

VHDL

If you are using Cadence NC-VHDL:

```
success := mmtcleval("set memid [mminstanceid :testbench:top:sdram]");  
success := mmtcleval("mmreadword $memid 0");
```

If you are using Mentor Graphics ModelSim:

```
success := mmtcleval("set memid [mminstanceid /testbench/top/sdram]");  
success := mmtcleval("mmreadword $memid 0");
```

mmtclcallback - This command is used to call a Tcl function when an assertion “fires”. Assertions are discussed in “Setting Assertions on Memory Transactions” on page 82. This command will return a “0” value if successful and a “-1” value if an error was encountered.

Example:

Use a Tcl callback to call another Tcl procedure (“Func1”) that is in the Tcl script “functions.tcl” when an assertion (bkpt) “fires”. (Assertions are discussed in “Setting Assertions on Memory Transactions” on page 82).

Verilog

```
success = $mmtcleval("source functions.tcl");  
success = $mmtclcallback(memid, bkpt, "Func1");
```

VHDL

```
success := mmtcleval("source functions.tcl");  
success := mmtclcallback(memid, bkpt, "Func1");
```

Tcl

```
source functions.tcl  
mmtclcallback memid bkpt Func1
```

Note that you must also use the *mmtcleval* command from your Verilog or VHDL testbench to source the Tcl handler script (functions.tcl). This Tcl handler function also expects the following parameters:

- bkpid = break-point id of the assertion

- instid = instance id
- addr = associated address for the assertion
- bkptype = assertion type(assert_datavalue,assert_access,assert_parity)
- access = type of access(read,write,etc)
- data = associated data for the above mentioned address
- mask = masked bits if any
- compare = compare string if any

An example of this Tcl callback handler would look like this:

```
proc Func1 {bkpid instid addr bkptype access data mask compare}
{
  set curtime [mmttime] //displays the time of assertion
  puts "$curtime:$bkpid $instid $addr $bkptype $access $data $mask
$compare"
  mmbreak
}
```

4.4.3.1 Tcl Callback Helper Commands

Denali has added some callback “helper” commands that can be called from within an assertion callback function. These are described below.

mmgetids - This command will return the memory id’s used in the design. These are integer values.

Tcl

`mmgetids` - will return {0} for a design with a single memory instance

mmgetinfo***byid*** - This command will return the width (in bits) of the memory instance referenced, the number of address locations, the instance name, and the type of memory.

Tcl

`mmgetinfo`***byid*** 0 - Returns: {9 2097152 testbench.uut1 sdram}

Where ‘9’ is the width of the device (in bits), 2097152 is the decimal number of address locations, testbench.uut is the instance name, and the memory type is an SDRAM.

mmnote - This command prints information about an assertion that has occurred. It can be called inside an assertion callback function.

Tcl

`mmnote` “This is a callback note”

mmbreak - This Tcl command stops the simulation. This would normally be called inside an assertion callback function.

Tcl

```
mmbreak
```

mmexit - This Tcl command will exit the simulation. This would normally be called inside an assertion callback function.

Tcl

```
mmexit
```

mmtime: This Tcl command gets the current time of simulation. Returns the time as a string. This would normally be called inside an assertion callback function.

Tcl

```
mmtime
```

4.4.4 Callback Commands

mmdisablecallbackall: This Tcl command is used to disable all callbacks setup in the Data Driven Verification API (DDV-API). This command can be called from Verilog or VHDL using the mmtcleval command.

Verilog

```
success = $mmtcleval("mmdisablecallbackall");
```

VHDL

```
success := mmtcleval("mmdisablecallbackall");
```

Tcl

```
mmdisablecallbackall
```

mmenablecallback: This Tcl command is used to enable callbacks on specific memory instances setup in the Data Driven Verification API (DDV-API). This command can be called from Verilog or VHDL using the mmtcleval command. NOTE-since all callbacks are enabled by default, to enable a single callback, you must use mmdisablecallbackall prior to enabling individual callbacks.

Verilog

```
success = $mmtcleval("set mem_id [mminstanceid tb.mem.sdram0 ]");  
success = $mmtcleval("mmenablecallback $mem_id");
```

VHDL

If you are using Cadence NC-VHDL:

```
success = $mmtcleval("set mem_id [mminstanceid :to:mem:sdram0]");  
success = $mmtcleval(" mmenablecallback $mem_id");
```

If you are using Mentor Graphics ModelSim:

```
success = $mmtcleval("set mem_id [mminstanceid /to/mem/sdram0]");  
success = $mmtcleval(" mmenablecallback $mem_id");
```

Tcl

```
set mem_id [mminstanceid tb.mem.sdram0]  
mmenablecallback $mem_id
```

mmdisablecallback: This Tcl command is used to disable callbacks on specific memory instances setup in the Data Driven Verification API (DDV-API). This command can be called from Verilog or VHDL using the mmtcleval command.

Verilog

```
success = $mmtcleval("set mem_id [mminstanceid tb.mem.sdram0 ]");  
success = $mmtcleval("mmenablecallback $mem_id");
```

VHDL

If you are using Cadence NC-VHDL:

```
success = $mmtcleval("set mem_id [mminstanceid :to:mem:sdram0]");  
success = $mmtcleval(" mmenablecallback $mem_id");
```

If you are using Mentor Graphics ModelSim:

```
success = $mmtcleval("set mem_id [mminstanceid /to/mem/sdram0]");  
success = $mmtcleval(" mmenablecallback $mem_id");
```

Tcl

```
set mem_id [mminstanceid tb.mem.sdram0]  
mmdisablecallback $mem_id
```

mmsetaccesscallbackmask: This Tcl command is used to enable callbacks based on the write mask field for specific memory instances setup in the Data Driven Verification API (DDV-API). The callback is triggered only when the write mask matches the string specified. This command can be called from Verilog or VHDL using the mmtcleval command.

Verilog

```
success = $mmtcleval("set mem_id [mminstanceid tb.mem.sdram0 ]");  
success = $mmtcleval("mmsetaccesscallbackmask $mem_id hff00");
```

VHDL

```
success = $mmtcleval("set mem_id [mminstanceid :to:mem:sdram0]");
success := mmtcleval("mmsetaccesscallbackmask $mem_id hff00");
```

```
success = $mmtclevl("set mem_id [mminstanceid /to/mem/sdram0"]);
success := mmtclevl("mmsetaccesscallbackmask $mem_id hff00");
```

```
set mem_id [mminstanceid tb.mem.sdram0]
mmsetaccesscallbackmask $mem_id 'hff00
```

```
InitialMemoryValue 0x3018
```

This string cannot expand to be longer than the memory's word width (typically the data size). If shorter, the unspecified bits will be filled with 0s.

Any other value will initialize memory to all 'X's. You may also generate random data for the initial data by specifying:

- InitialMemoryValue randomNoUpdate - for new random data after multiple reads.
- InitialMemoryValue randomWithUpdate - for the same random data after multiple reads.

NOTE: *For logically addressed memory if you are using **InitialMemoryValue randomWithUpdate**, you will get write callbacks if write callbacks are enabled.*

For details, refer to [“Controlling Your Memory Simulation Models - The .denalirc File” on page 39](#).

Alternatively, you can use any arbitrary C function to specify the initial data, including random data, random data with parity, etc.

Refer to \$DENALI/ddvapi/example/fillValue for an example.

- Use the “**mmsetfillvalue**” command in your simulation testbench.

NOTE: ***mmsetfillvalue** only sets the value for the unwritten locations. It does not reset them. To reset the memory locations, use **\$mmreset**. For details, refer to “Resetting the Memory Contents” on page 67.*

The size of the fill value is the width of the memory device. Use hexadecimal format “0x” to define the actual fill value. Examples for setting the fill value of 0x55 for an 8 bit wide SDRAM instance “tb.mem.sdrām0” in Verilog and VHDL are below.

An optional 3rd argument has recently been added to **mmsetfillvalue**. This argument is used for suppressing any informational messages. If the value of the third argument is 1, any informational messages are not displayed. If the value is 0 or no value is provided as a third argument, all messages are displayed as usual.

This command will return a “0” value if successful and a “-1” value if an error was encountered.

Verilog

```
status = $mmsetfillvalue("tb.mem.sdrām0", "0x55", "1"); //suppress  
the informational messages
```

VHDL

If you are using Cadence NC-VHDL:

```
status := mmsetfillvalue("<instance_id>", ":tb:mem:sdrām0", "0x55");  /  
/No message suppression (default)
```

If you are using Mentor Graphics ModelSim:

```
status := mmsetfillvalue("<instance_id>", "/tb/mem/sdrām0", "0x55");  /  
/No message suppression (default)
```

Tcl

```
mmsetfillvalue /tb/mem/sdram0 0x55 1 # supress messages
```

NOTE: *mmsetfillvalue ONLY pertains to physical memories. This command cannot be used to set the fill value of a logically addressed memory. To accomplish this, you must use mmsetfillvalue on each physical memory instance that is part of the logically addressed memory.*

4.5.3 Loading Memories From a File

All Denali memory models can be loaded from a file. The file load command takes two arguments. The first is the instance name for the Denali model you wish to load and the second is the file which contains the address and data values. The width of the data values in the load file must match the width of the device you are attempting to load. The syntax of the file is as follows:

```
<start_addr>:<end_addr>/<data>;
```

or

```
<addr>/<data>;
```

All address and data values are in HEX format. The load file can contain blank lines and the comment character is “#”. Example:

```
# File name: load.dat
0/21;
1/22;
3/23;
4:1F/55;
20:1FFF/FF;
# End of load.dat
```

Below are examples of the load command:

Verilog

```
status = $mmload("tb.mem.sdram0","load.dat");
```

VHDL

If you are using Cadence NC-VHDL:

```
status := mmload("/:tb:mem:sdram0","load.dat");
```

If you are using Mentor Graphics ModelSim:

```
status := mmload("/tb/mem/sdram0","load.dat");
```

Tcl

```
mmload tb.mem.sdram0 load.dat    # Verilog
mmload /tb/mem/sdram0 load.dat  # VHDL
```

NOTE: *You can also specify a file to be loaded upon model initialization. This is done in the HDL shell file that is created. For details, refer to “Creating an HDL Shell with the PureView GUI” on page 23.*

4.5.4 Memory Content File Format

Memory format files are used to Save, Load, and Compare the contents of a memory instance using standard text files.

Each memory format file is composed of a set of records. Each record must be written within one line of ascii text, but multiple records may be contained on one line. Use the following syntax when composing these files:

```
start_addr [:|- end_addr] / data;
start_addr / data [data ...];
[# comment]
```

“[]” indicates an optional component of the record. “|” indicates the logical OR. For the record specification above, this indicates that either character “:” or “-” may be used to separate the end_addr from the start_addr.

Comments are indicated by the character “#” and are continued until the end of the line. A line can contain just a comment.

Base Specifier Prefix

The start_addr and end_addr for the mmload and mmcomp commands and the data can be specified in binary, octal, hexadecimal, or decimal format by using a base specifier prefix. By default, if no base specifier is used, the base is hexadecimal.

Use the following prefixes as base specifiers:

```
'b - binary
'o - octal
'd - decimal
'h - hexadecimal
```

The start_addr is a required component of a memory format record. When used with an end_addr, the record specifies the value for a contiguous, inclusive address range. When used alone, the record specifies the memory contents of a single address if only one data value is supplied. If multiple data values are supplied, then the record specifies the memory contents sequentially beginning with the address start_addr.

Use the end_addr to specify the end of an address range.

Use either the character “:” or “-” to separate the start_addr from the end_addr.

For the mmsave and mmsaverange commands, the format of the output data can be set only as hex. In case you want to change this format setting, you can use SaveDataRadix and SaveAddressRadix .denalirc variables. These can be set to 2 (binary), 8 (octal), 10 (decimal), or 16 (hexadecimal), depending on which "radix" you want the output of the saved files. Alternatively, you can dynamically set these from your testbench. The default value is 16.

For example, to change the address radix dynamically to binary, you can use:

```
success = $mmtcleval("mmsetvar SaveAddressRadix 2");
```

For example, to change the data radix dynamically to binary, you can use:

```
success = $mmtcleval("mmsetvar SaveDataRadix 2");
```

As with the address(es), by default the data and address is in hexadecimal. Unknowns are allowed and are indicated by either the character “X” or “x”. However, unknowns cannot be used with decimal values.

While it is good style and practice to specify the data with a width equivalent to the data width of the memory instance, this is not required. Data will be filled with leading 0's if the value is smaller than the data width of the memory instance.

If the value is too large, the format file will be invalid for the relevant instance.

Use the character “;” to indicate the end of a memory format record. This is not necessary for the last record in the file. Multiple records may be entered on one line of text.

Converting Motorola S-record and Intel Hex Format Files

Denali has Tcl scripts that can convert a Motorola S-record format file or an Intel hex format file into a Denali load file. Contact support@denalisoftware.com if you would like these Tcl scripts.

Parity Bits

Certain memory types (primarily SRAM devices) have optional parity bits specified in the memory model. The number of bits used for parity is determined by the SOMA file setting for a particular device. Normal devices use a single bit of parity to cover 8 bits of data. The parity bits are ALWAYS the least significant bits of the data field that is to be loaded. For example, if an 8 data bit + 1 parity bit SRAM device is used, the data bits will be from [8:1] and the parity bit will be bit [0]. For a 16 data bit + 2 parity bit device, the data bits will be from [17:2] and the parity bits will be [1:0]; where bit [1] covers data bits [17:10] and bit [0] covers data bits [9:2]. You must consider these when loading the data into these devices.

Examples

Example 1:

The following example loads the hexadecimal value “70FF” into the address at hexadecimal “AB73”.

```
AB73 / 70FF;
```

Example 2:

The following example loads hexadecimal values into the addresses beginning with the address at hexadecimal “AB73” (and ending with the hexadecimal address “AB7A”).

```
AB73 / 70FF 7100 7101 7102 7103 7104 7105 7106;
```

Example 3:

The following examples both load the binary value “1100XXXX” (with unknowns) into the address range starting with the initial address and ending with the decimal address “2478”. Note that you can use either the “:” or the “-” address range separator.

```
0: 'd2478/ 'b1100XXXX;  
0- 'd2478/ 'b1100XXXX;
```

Example 4:

The following example shows the ability to specify multiple records on one line and illustrates ending a line with a comment.

```
0/00; 1/01; 2/02; 3/03; # pattern loading
```

4.6 Specifying Memory Instances

Denali MMAV-2001 references memories in the design by using the HDL instance name of the model in the design. Denali provides two functions, “*mminstanceid*” for Verilog and “*mmgetinstanceid*” for VHDL. These commands can be used to extract the instance name to reference a specific memory device for all of the Denali memory model commands.

The typical usage is to assign a variable or an integer to the specified instance name. This variable can then be used as a “shortcut” when referencing Denali memory models.

There are two forms of the commands. The first takes the full instance path name as the memory instance name, while the second takes a relative path. If the user knows the full hierarchical path down to the memory model, then this is the preferred method.

If the user needs more portability and is not sure of the higher hierarchy levels, then the user can use a relative path to identify the memory model within a particular module or architecture block. The relative path is **ONLY** valid if *mminstanceid* or *mmgetinstanceid* is used within a module or architecture block. The resulting variable or integer is now associated with that instance and can be used with other commands.

In VHDL, however, there is another issue. When instantiating models using a **GENERATE** statement, a special form of *mmgetinstanceid* is needed. A “path_name” parameter is needed. This allows Denali to extract the full path name from the **GENERATE** block to form the correct instance name.

Example (Full hierarchical instance name):

Verilog

```
memory_id0 = $mminstanceid("testbench.top.device0");
```

VHDL

If you are using Cadence NC-VHDL:

```
memory_id0 := mmgetinstanceid(":testbench:top:device0");
```

If you are using Mentor Graphics ModelSim:

```
memory_id0 := mmgetinstanceid("/testbench/top/device0");
```

Example (Relative instance name, relative to module of instantiation):

Verilog

```
memory_id0 = $mminstanceid("device0");
```

VHDL

```
memory_id0 := mmgetinstanceid("device0");
```

Example (Relative instance name used in **GENERATE** statements):

VHDL

```
memory_id0 := mmgetinstanceid(device0'path_name);
```

4.7 Resetting the Memory Contents

At any point during simulation, you can reset the memory contents of the specified memory instance back to its initial state. The *mmreset* command accomplishes this.

Verilog

```
success = $mmreset ("tb.mem.sdram0");
```

VHDL

```
success := $mmreset ("tb.mem.sdram0");
```

Tcl

```
mmreset tb.mem.sdram0
```

4.8 Reading and Writing Memories

MMAV enables you to read and write any memory location from the test bench. These commands can be issued at any time during the simulation. You must specify the instance you wish to read or write as well as the address or addresses. There are multiple read and write commands for Verilog.

The Verilog commands for reading or writing memories can take explicit instance names or implicit instance ids obtained from the “*mminstanceid*” commands. The difference between the two is that the implicit instance id commands are faster than the explicit instance names. Use the instance id commands if you are reading and writing memories in the test bench often. You can also read and write contiguous locations in one command by specifying the starting address and the number of locations to read or write. The syntax of the commands are listed below. This command returns a “0” value if successful and a “-1” value if an error was encountered.

NOTE: *Certain SRAM devices have parity bits associated with the data bits. Refer Figure , “Parity Bits,” on page 65 for details on handling these parity bits.*

Also note that the address input must be a string value (“0x00”) when using mmreadword and in verilog format (‘h00) when using mmreadword2 or mmreadword3.

```
$mmreadword("<instance name>", "<address>", <param>);  
$mmreadword2(<instance id>, <address>, <param>);  
$mmreadword3(<instance id>, <address>, <number>, <param>, ..., <param>);
```

Verilog

Examples:

```
status = $mmreadword("tb.mem.sdram0", "0x000a", tmp_read);  
  
m_id = $mminstanceid("tb.mem.sdram0");  
status = $mmreadword2(m_id, 'h200, tmp_read);  
status = $mmreadword3(m_id, 'h200, 2, tmp_read, tmp_read2);
```

The Verilog write commands have various syntax based on whether the data you wish to write is explicit to the commands or resides in a register or parameter in the simulation.

Choose the correct command for your application. The syntax for these commands are as follows.

NOTE: *The data value “<value>” must be a binary string value. When you are using the mmwriteword, the address input must be a string value (“0x00”). When you are using mmwriteword2-5, the address input must be in Verilog format (‘h00) or a variable.*

Examples:

```
$mmwriteword("<instance name>", "<addr>", "<value>");  
$mmwriteword2(<instance id>, <addr>, <value>);  
$mmwriteword3(<instance id>, <addr>, <num>, <value>, ..., <value>);  
$mmwriteword4(<instance id>, <addr>, <param>);  
$mmwriteword5(<instance id>, <addr>, <num>, <param>, ..., <param>);
```

Examples:

```
status = $mmwriteword("tb.mem.sdram0", "0x200", "01010101");  
  
m_id = $mminstanceid("tb.mem.sdram0");  
  
status = $mmwriteword2(m_id, 'h200, "01010101");  
status = $mmwriteword3(m_id, 'h200, 2, "01010101", "10101010");  
status = $mmwriteword4(m_id, 'h200, tmp_write);  
status = $mmwriteword5(m_id, 'h200, 2, tmp_write, tmp_write2);
```

For VHDL and Tcl, there are currently only two commands for reading and writing memory locations. The syntax is as follows. The status of the command is returned in the <status> variable. The return value of TRUE means success and the value of FALSE means failure to write the data.

VHDL

```
mmreadword(<instance id>, <addr>, <param>, <status>);  
mmwriteword(<instance id>, <addr>, <data>, <status>);
```

Example (if you are using Cadence NC-VHDL):

```
m_id := mmgetinstanceid(":tb:mem:sdram0");  
mmreadword(m_id, 200, tmp_read, status);  
mmwriteword(m_id, 200, tmp_write, status);
```

Example (if you are using Mentor Graphics ModelSim):

```
m_id := mmgetinstanceid("/tb/mem/sdram0");  
mmreadword(m_id, 200, tmp_read, status);  
mmwriteword(m_id, 200, tmp_write, status);
```

Tcl

```
set <param> [mmreadword <instance id> <addr>]
```

```
mmwriteword <instance id> <addr> <data>
```

Examples:

```
set tmp_read [mmreadword /tb/mem/sdram0 200]  
mmwriteword /tb/mem/sdram0 200 10100101
```

4.8.1 Masked Memory Writes

Verilog

```
$mmwritewordmasked (<instance_id>, <address>, "value", "mask")
```

VHDL

```
mmwritewordmasked (<instance_id>, <address>, "value", "mask", status)
```

This Verilog and VHDL-only function is the same as the Verilog `$mmwriteword2` function with the difference being that only the bit locations set to “1” in the mask are copied from “value” (bit-string) into the instance id `<instance_id>` at address `<address>`. The remaining bit locations at the instance id `<instance_id>` at address `<address>` are preserved to their previous bit-value. This function returns 0 when successful.

Verilog

```
i2 = $mminstanceid("testbench.uut1");  
success = $mmwriteword2(i2, 'h14, "00001111");  
success = $mmwritewordmasked(i2, 'h14, "01010101", "00110011");
```

This example would result in the value "00011101" being at address 'h14 at instance “i2”.

VHDL

```
i2 := mmgetinstanceid("testbench.uut1");  
mmwritewordmasked(i2, 0, "01010101", "00110011", status);
```

Tcl

Not available

4.9 Saving and Comparing Memory Contents

All Denali memories allow you to save the contents of the memories to a file or compare the memory contents to that file.

There are three commands for these features - `mmsave`, `mmsaverange`, and `mmcomp`. The first two allow you to save the entire contents or only a specific range of addresses. The third command allows you to compare the entire contents of memory to a specified file. For the `mmsave` and `mmsaverange` commands, the format of the output data can be set

only as hex. The file format for mmcomp command is described in Section 4.5.4, “Memory Content File Format,” on page 64.

The Verilog and VHDL commands are described as follows. This command will return a “0” value if successful and a “-1” value if an error was encountered.

Verilog

```
$mmsave("<instance name>","<file name>");  
$mmsaverange("<instance name>","<file_name>",<start addr>,<end addr>);  
$mmcomp("<instance name>","<file name>");
```

Example:

```
status = $mmsave("tb.mem.sdram0","save.dat");  
status = $mmsaverange("tb.mem.sdram0","save.dat",'h0','h1f');  
status = $mmcomp("tb.mem.sdram0","save.dat");
```

VHDL

```
mmsave("<instance name>","<file name>");  
mmsaverange("<instance name>","<file name>",<start addr>,<end addr>);  
mmcomp("<instance name>","<file name>");
```

Example:

```
status := mmsave("/tb/mem/sdram0","save.dat");  
status := mmsaverange("/tb/mem/sdram0","save.dat",0,31);  
status := mmcomp("/tb/mem/sdram0","save.dat");
```

Tcl

```
mmsave <instance name> <file name>  
mmsaverange <instance name> <file name> <start addr> <end addr>  
mmcomp <instance name> <file name>
```

Example:

```
mmsave /tb/mem/sdram0 save.dat  
mmsaverange /tb/mem/sdram0 save.dat 0 31  
mmcomp /tb/mem/sdram0 save.dat
```

The mmcomp command enables you to go through the file that you specify and determines if any address/data pair listed there mismatches with the memory you give it to compare to.

NOTE: *The mmcomp command does NOT do the other way around (i.e. loop through the entire memory and compare to the file). So if you write to an address outside of those in the file, it does not catch it as it is not comparing in that direction.*

As a workaround, you can put lines in the file that do not have a specific value, and set to the initial value. In order to do this, after you run mmload with load.dat file,

save it immediately with mmsave as comp.dat. This automatically fills in the initial values everywhere you did not specify the address. Then, at a later point, you can use mmcomp command and compare it against comp.dat.

NOTE: *mmSave is not directly available in VHPI. You can use the mmtclevel as an alternative solution.*

NOTE: *In system memory, when the data written to any location matches the "fill value" of the memory, it does not dump that data for mmsave and mmsaverange.*

4.10 Recalculating Clock Cycle Time

Certain memory classes allow the option to recalculate the clock cycle time. With this feature enabled, the Denali memory models will recalculate the clock cycle on the next rising edge of the clock. This is useful if you have “IrregularClock = 0” (see “IrregularClock” on page 43) as certain timing parameters which are based on the clock may be incorrect if the Denali model is not notified of the clock cycle time change.

```
mmrecalculatecycle( "instance_name" )
```

Verilog

```
success = $mmrecalculatecycle( "testbench.uut1" );
```

VHDL

```
success := mmtclevel("mmrecalculatecycle testbench.uut1");
```

Tcl

```
mmrecalculatecycle testbench.uut1
```

4.11 Re-loading SOMA Files and Changing Timing Parameters on-the-fly During Simulation

Starting in version 3.00, two new features are added for a subset of our memory classes. These features allow you to dynamically alter SOMA file parameters or even load in a completely different SOMA file than is referenced in the HDL wrapper. Currently these features are valid for the following memory classes:

- DDR3-SDRAM
- DDR2-SDRAM
- DDR-SDRAM
- FLASH-AMD
- FLASH-INTEL

- FLASH-NAND
- FLASH-ONE-NAND
- GDDR3
- GDDR4
- SDRAM
- RDRAM
- FCRAM
- RDRAM
- QDR-SSRAM
- NVM_DDR
- MS
- MS-PRO

1. **mmsomaset** (“<instance_name>”, “<parameter_name>”, “<parameter_value>”, “<parameter_units>”);

Use this if you just want to change a small number (<3) timing numbers from the original SOMA file. If you want to change more than that, it's more efficient to use *mmsomaload* as described below. This can be called via the pli (*\$mmsomaset*) or via mmtcleval.

Simulation continues from that point with the new numbers. Memory contents are unaffected. Note that both parameter_value and parameter_units fields are required.

Note that the **parameter_units** can be “ns” for nano-seconds, and “clk” for time units of clocks as specified in the original SOMA file.

Examples:

Verilog

```
success = $mmsomaset ( "tb.mem0", "toh", "15.0", "ns" );
```

VHDL

Use mmtcleval with the Tcl command below.

If you are using Cadence NC-VHDL:

```
ex. mmtcleval "mmsomaset :tb:mem0 toh 15 ns";
```

If you are using Mentor Graphics ModelSim:

```
ex. mmtcleval "mmsomaset /tb/mem0 toh 15 ns";
```

Tcl

```
mmsomaset /tb/mem0 toh 15 ns
```

2. **mmsomaload** (“<instance_name>”, “<file_name>”);

Use this to change multiple timing numbers, and/or to change pin widths. This is more efficient for changing multiple timing numbers than a series of *mmsomaset* commands because many of the checks will only get done once, and recalculations of things based on clks etc. will only happen once. Note that though this SOMA file will need to have the things you want to change in it, it must be a complete SOMA file.

Once *mmsomaload* is issued, the memory is reallocated and all contents are thrown away. You will see warnings if you resize pins and they don't match what the simulator has. You will get a fatal error if the size of any signal is larger than what the simulator has, which is the maximum size that you can have for any pin (this width is the width from the Verilog or VHDL shell).

An init file is NOT read (as it probably doesn't apply now) and you have to go through all the steps you would expect with the brand new memory. However if you have other memories in your design they are not affected. This is basically “starting over” with that memory, except simulation time continues from where it left off and you don't have to exit the simulator, recompile, etc.

Implementation Note: If you are only interested in changing timing and do not want the memory reset, you should do a series of *mmsomaset* commands instead.

Restrictions:

- Cannot change classes with this technique. Once a DDR memory, always a DDR memory.
- Cannot make pin widths any bigger than the original HDL shell. You can make them smaller, then grow them again up to the original HDL shell width.

Examples:

Verilog

```
success = $mmsomaload ( "tb.mem0", "new_soma.spc" );
```

VHDL

Use mmtcleval with the Tcl command below.

If you are using Cadence NC-VHDL:

```
mmtcleval "mmsomaload :tb:mem0 new_soma.spc";
```

If you are using Mentor Graphics ModelSim:

```
mmtcleval "mmsomaload /tb/mem0 new_soma.spc";
```

Tcl

```
mmsomaload tb.mem0 new_soma.spc
```

4.12 Error Message Control

Starting in Denali version 2.800-0018, an additional Tcl command was added to provide the user with the flexibility to turn on/off error reporting. Two new commands called “*mmerrormessageson*” and “*mmerrormessagesoff*” were added. The functions can be called at any time to turn on/off error messaging from the Denali models. By default the messaging is always on.

Verilog

```
success = $mmtcleval("mmerrormessageson");  
success=$mmtcleval("mmerrormessagesoff");
```

VHDL

Use mmtcleval with the Tcl command below.

```
mmtcleval "mmerrormessageson";  
mmtcleval "mmerrormessagesoff";
```

Tcl

```
mmerrormessageson  
mmerrormessagesoff
```

In addition to the above commands, there are two commands that can be used to turn on/off the history and tracefile generation dynamically during simulation:

\$mmdebugon;

Allows you to create a window of history and trace file information. This command is overridden when the HistoryFile and TraceFile options are enabled in the *.denalirc* file. Returns 0 when successful.

Verilog

```
initial #5000 success = $mmdebugon;
```

VHDL

```
wait for 5000 ns;  
success := mmdebugon;
```

\$mmdebugoff;

Closes the window of history/trace information. Returns 0 when successful.

Verilog

```
initial begin
    #50000 success = $mmdebugon;
    #50000 success = $mmdebugoff;
end
```

VHDL

```
wait for 50000 ns;
    success := mmdebugon;
wait for 50000 ns;
    success := mmdebugoff;
end
```

4.13 Forcing Clock Cycle Recalculation

If the memory class supports this capability, MMAV recalculates the clock cycle on the next rising edge of the clock. This is useful if IrregularClock = 0, but somewhere along the line the clock cycle changes. If you do not notify the model, the timing parameters which are based on the clock may be incorrect.

Verilog

```
$mmrecalculatecycle( "instance_name" )
```

4.14 RDRAM (Rambus) Specific Model Considerations

4.14.1 Turbo Channel Model for RAMBUS

Overview

In addition to high-performance models for Rambus discrete Direct-RDRAM components, Denali offers a “Turbo Channel Model” which delivers the highest performance of any Direct RDRAM channel simulation model on the market. This option, using algorithms for accelerating the simulation performance, the Turbo Channel Model “collapses” the individual RDRAM instances into a single model object without compromising accuracy. In most cases, the simulation performance shows “n” times improvement over discrete RDRAM models where “n” is the number of RDRAM components on the Rambus Channel being simulated.

Licensing

An additional license, *Denali_SIM_rdrdram_turbo*, is required for simulation of the RDRAM Turbo Channel Model. Contact Denali Software at: info@denalisoft.com to inquire about this option.

Device Numbers

To use the RDRAM Turbo model, the user needs to specify the generic parameter: *device_number*. This string parameter defines the number of devices on a Rambus channel. At the HDL level, one instance represents one channel; subsequently, individual devices should not be instantiated in the test bench.

The devices on the channel will have names in the form:

```
<channel_name>_d%d
```

For example, if the channel instance name is *ch1* and it has a total of 8 devices, then the devices would be named as:

```
ch1_d0, ch1_d1, ch1_d2, ch1_d3, ch1_d4, ch1_d5, ch1_d6, ch1_d7
```

This naming convention should be followed when loading the memory contents with the *mmload* function.

Note: If the generic *device_number* is undefined or equals to 0, then the Turbo mode is turned off. This means the instance is treated as a single device instance instead of as a channel instance. In this case, the user has to instantiate all instances on the channel. The serial pins need to be connected manually also.

Device id's

In Turbo Mode, the model automatically sets device id's to 0...n-1. No *mmload* is needed anymore. However, *checkInitialization* has to be deselected to enable default id.

Channel Inversion

Some RAC models (and/or test benches) drive ROW/COL/DQA/DQB with physical voltage levels. If the parameter *invertChannel* is set in PureView (in the SOMA file), the resulting model will internally invert the logic signals ROW/COL/DQA/DQB.

Channel Delay Settings

The Denali RDRAM Turbo Channel Model stores channel delay settings in the unused RDRAM control register *0xf*. The last three bits of *0xf* are used to specify the Denali chan-

nel interconnect delay in terms of CFM/CTM cycles, ranging from 0 to 7 cycles. For Example, the parameter:

```
0f/'b00000000000000110; channel delay=6cyc
```

would cause a channel interconnect delay to be added to the register TPARM and TCDLY1 values resulting in a new tCAC read data delay as follows:

$$tCAC = channel_delay + 3 * tCYCLE + tCLS_C + tCDLY0_C + tCDLY1_C$$

Notice that channel delay in the 0xf register is specified as number of CFM/CTM cycles (not half-cycles) because it is the from/to master round trip delay.

.denalirc Options

- **WarnSuppress**

In addition to normal timing and protocol error conditions, the Denali RDRAM model issues warnings for the following hazardous operations:

- overwriting the write buffer before it's retired (e.g. WR-WR-RD-RD-RTR).
- precharging a bank before it's retired (rule CR8).

These warnings can be suppressed by setting the *WarnSuppress* parameter in the *.denalirc* file as follows:

```
WarnSuppress 1
```

- **TimingChecksReportOnly**

Turning this *.denalirc* option on will disable the Denali model feature that corrupts memory and drives “X’s” on the data-bus when there are timing errors. This option allows timing errors to only cause the model to issue messages and will NOT drive “X” when seeing timing errors. This can be very useful in early evaluation of errors, but will allow the simulation to continue, though reporting errors.

```
TimingChecksReportOnly 1
```

- **TimingChecksStartTime**

Denali also provides the capability to turn on timing checks at a specific time. This allows time for reset and device initialization before Denali models start checking for timing errors. See examples below for syntax.

```
TimingChecksStartTime 0ps
TimingChecksStartTime "20 us"
TimingChecksStartTime "200ms"
TimingChecksStartTime 200000 ns
```

Device Refresh Options

All Denali memory models use a sophisticated dynamic memory allocation algorithm to minimize the memory footprint during simulation. Memory requirements for the RDRAM Turbo Channel Model can be further reduced by specifying refresh checking in a single

device, as opposed to all devices in the channel. The generic parameter *device_number* is used to specify refresh intervals for a specific device on the channel as follows:

```
device_number = "32;RefreshCheckOneDev:1"
```

The above example illustrates the instantiation of a channel with 32 devices, refresh checking one device only.

Note: Refresh commands are normally issued through channel broadcast, so checking only one device should preserve good model accuracy.

Multi-Bank Refresh Options

Multi-bank refreshes are controlled by the *numBanksPerRefresh* parameter. If multiple banks are refreshed for each refresh command, then the banks share the same lower address bits. For example:

```
numBanksPerRefresh 2
```

If the setting above was used with a 16 bank device, the first refresh would occur in banks 0 & 8. The next refresh would occur in banks 1 & 9, and so on.

Turbo Mode - Support for Different Sized Devices on the Same Channel

Previously, all devices on the same channel shared the same SOMA file, meaning that they all had to be the same configuration. The generic parameter, “**device_number**”, has been expanded to support different sized devices on the channel. For example:

Eight 128M devices plus sixteen 64M devices on a single channel would get instantiated as:

```
device_number => "24;8:128M_800.spc;16:64M_800.spc";
```

Note that **memory_spec** has no effect any more.

Denali RDRAM Byte Ordering Options

The Denali RDRAM model handles the dualoct in the following fashion in memory (for file loads and saves):

DQA[8]...DQA[0]	DQB[8]...DQB[0]	DQA[8]...DQA[0]	DQB[8]...DQB[0]
\.....cfm clk 0			\..... cfm clk 7	
0	71		72	143

This results in bit ordering as such:

data[0] is DQA[8] of tick 0, data [143] is DQB[0] of tick 7

The mask array is also constructed using the same byte order and when a dualoct gets read out, the data is presented in the same order.

This byte ordering is different from RMC/RAC. In RMC/RAC, a dualoct is split into DQA/DQB as follows:

DQB(clk 7) ... DQB(clk 0)	DQA(clk 7) ... DQA(clk 0)
\-----/	\...../
143 72	71 0

If you wish to use the RMC/RAC byte ordering for comparison or for loading a memory content file, an environment variable in *.denalirc* can be set to allow this configuration.

RdramRMCByteOrder 1

In this case, the Denali RDRAM model will save/load the dualoct in the RMC/RAC byte order as above.

5 MMAV Special Verification Features

Denali has added a suite of additional verification functions that allow you to add verification checks to your testbenches and also organize physical memories into logical memory views. These features are intended to be a part of your regression tests and help to verify that your entire system is functioning properly. You can verify many parts of your system by tracking the memory transactions as they occur in your simulations. An example of this is a system that handles telecom packets. The packets come in from a source and are written into memory. These packets are then read from memory and modified and sent out again. One of the properties of this system is that any packet written into memory at a particular address must be read out before another packet can be written into that memory location. One verification feature to assist in checking this system are assertions that checks that any location in memory is written twice in a row. This assertion monitors all memory accesses to a certain memory and will report an error when a location is written for the second time without a read in between. By using this feature, you will be alerted immediately when the system violates this packet protocol.

MMAV can also create logically addressed views of physical memory components. These logical views are important to debugging and test bench generation and support. An example of a logically addressed view is collecting together multiple SDRAM devices into a single bank of memory. A logical 32 bit bank of SDRAM memory can be created from four eight bit SDRAM physical instances. Once you have created the logical memory view, you can load, store, compare, read and write this memory view and the MMAV functions will automatically access the correct physical instances.

Another verification feature in MMAV is Transaction Lists. A transaction list allows users to verify the sequence of read and write operations expected on a memory segment. The read/write operations, their addresses and (optionally) their values are added in sequence to a transaction list. Memory segments that are added to the transaction list are monitored to check if their read/write sequence matches the read/write operations registered with the transaction list. If the sequence, address and/or values do not match the list of registered transactions, an assertion is triggered and one of three actions can be taken: a notification can be issued, or simulation can be stopped with a message, or simulation can exit with a message. Each registered read/write can be matched against a single read/write or against N ($N > 0$) read/writes. This can be controlled on a per operation basis.

5.1 Setting Assertions on Memory Transactions

To set an assertion to track memory references, you have to execute a command in the simulation test bench to register the assertion. The command returns a unique number that can be used to disable and enable the assertion throughout the simulation. Each command and its syntax is described in the following sections.

5.1.1 Memory Access Assertions

This assertion allows you to trigger an assertion when a particular memory is accessed in the specified manner. Typical uses for this assertion are to protect certain address ranges from unwanted accesses. Writing into address space that is designated as code as opposed to data can be registered so that an erroneous write to that address space triggers the assertion to stop the simulation. The syntax for this assertion is as follows.

```
mmassert_access(<instance id>,"<access>","<action>",<start address>,<end address>,[<address increment>]);
```

The allowable types for <access> are:

- **Read** - a read of a memory location.
- **Write** - a write to a memory location.
- **ReadorWrite** - either a read or a write.
- **ReadnoWrite** - a read of a memory location without a previous write to that location.
- **ReadRead** - two consecutive reads to the same memory location.
- **WriteWrite** - two consecutive writes to the same memory location.

NOTE: *The access types listed above are NOT case sensitive, they are shown this way for clarity.*

Action

The allowable values for <action> are:

- **Note** - a note in the simulation log is printed out and the simulation continues.
- **Break** - a note in the simulation log is printed and the simulation stops (pauses).
- **Exit** - a note in the simulation log is printed and the simulation exits.
- **Callback** (new in 3.2) - a verilog callback is generated when the assertion is triggered.

Start Address, End Address, Address Increment

These entries indicate the range of addresses to which the assertion is applied to. The address increment value allows for consecutive addresses within the starting and ending

address range (addr increment=1) or other selected addresses within the range (addr increment =/ 1).

If both a starting and ending address are given then the user may specify the additional parameter (optional) <address increment> which specifies the steps in which to increment the address within that particular address range. For instance if the starting address is specified as 20H and ending address as 30H and the <address increment> parameter is set to 2 it means that all the assertion checks will be done in steps of 2 from address locations 20H, 22H, 24H and so on until reaching the ending address of 30H. If no starting and ending address are specified then the check is done on the entire memory instance or if only the starting address is specified then the check is done from that particular address through the end of the address space.

Examples:

To halt the simulation whenever locations 0x0 through 0x4000 are written:

Verilog

```
m_id = $mminstanceid("tb.mem.sdram0");  
a_id = $mmassert_access(m_id,"write","break",'h0','h400','h1');
```

VHDL

```
m_id := mmgetinstanceid("/tb/mem/sdram0");  
a_id := mmAssert_Access(m_id,"write","break",0,400,1);
```

Tcl

```
set id [mminstanceid tb.mem.sdram0]  
mmassert_access $id write break 0 1024 1
```

(NOTE: again the address value (1024) is in decimal and not hexadecimal).

5.1.2 Data Access Assertions

You can also qualify any access assertion to add a data checking component. You can create an access assertion that will only fire based on the data that is being read or written. The triggering of the assertion is based on the comparison function of the data you specify against the actual data being read or written. You can also supply a mask value to check only certain bits of the data. The syntax for this command is as follows.

```
mmassert_datavalue(<memory id>,"<access>","<action>",<data  
value>,[<mask>], " <comparison>",<start address>,<end  
address>,<address increment>);
```

Comparison

Masking is performed on a bit by bit basis and comparisons are then performed on the resulting word. Allowable values are:

- **==** specifies that the accessed (masked) data is **equal to** the specified (masked) data
- **!=** specifies that the accessed (masked) data is **not equal to** the specified (masked) data
- **>** specifies that the accessed (masked) data is **greater than** the specified (masked) data
- **>=** specifies that the accessed (masked) data is **greater than or equal to** the specified (masked) data
- **<** specifies that the accessed (masked) data is **less than** the specified (masked) data
- **<=** specifies that the accessed (masked) data is **less than or equal to** the specified (masked) data

Data Value, Mask

This entry specifies the data and mask (in hex format) to be used for comparisons. A comma character is used to separate the optional mask value from the data value. If no mask is specified, all the bits are compared. If a mask value is given, only the masked portion of the data being accessed is compared to the data specified. If the mask bit is “1” for that data position, the comparison will take place. If the mask bit is “0”, the comparison for that data position will not be considered.

Start Address, End Address, Address Increment

These entries indicate the range of addresses to which the assertion is applied to. The address increment value allows for consecutive addresses within the starting and ending address range (addr increment=1) or other selected addresses within the range (addr increment =/ 1).

If both a starting and ending address are given then the user may specify the additional parameter (optional) <address increment> which specifies the steps in which to increment the address within that particular address range. For instance if the starting address is specified as 20H and ending address as 30H and the <address increment> parameter is set to 2 it means that all the assertion checks will be done in steps of 2 from address locations 20H, 22H, 24H and so on until reaching the ending address of 30H. If no starting and ending address are specified then the check is done on the entire memory instance or if only the starting address is specified then the check is done from that particular address through the end of the address space.

Example:

To set an assertion whenever the memory is written with values greater than 0x0 for addresses greater than 0x4000 to put a note in the simulation:

Verilog

```
m_id = $mminstanceid("tb.mem.sdram0");
a_id = $mmassert_datavalue(m_id,"write","note",'h0','h1f",>","'h400,
'h1ffff','h1');
```

VHDL

```
m_id := mminstanceid("/tb/mem/sdram0");
a_id :=
mmassert_datavalue(m_id,"write","note","0","00011111",>","1024,131071,1)
;
```

Tcl

```
set m_id [mminstanceid tb.mem.sdram0]
mmassert_datavalue $m_id write note 0 00011111 > 1024 131071 1
```

NOTE: again the address values (1024, 131071) are in decimal and not hexadecimal).

5.1.3 Global Memory Access Assertions

The previous assertion commands registered assertions on particular memory instances. The following assertions set the particular assertion on all memory instances in the test-bench. These assertions are

```
mmassert_uma("<action>");
```

This assertion will fire whenever the simulation attempts to read a location in memory that has not been written to or initialized by a load command.

```
mmassert_rwa("<action>");
```

This assertion will fire whenever a single location in memory has been written twice before a read has occurred.

```
mmassert_rra("<action>");
```

This assertion will fire whenever a single location in memory has been read twice before it is written.

Verilog

```
a_id = $mmassert_uma("break");
a_id = $mmassert_rwa("note");
a_id = $mmassert_rra("exit");
```

VHDL

```
a_id := mmassert_uma("break");
```

```
a_id := mmassert_rwa("note");
a_id := mmassert_rra("exit");
```

Tcl

```
mmassert_uma break
mmassert_rwa note
mmassert_rra exit
```

5.2 Parity Checking Assertions

If you have a protected memory with a parity bit, you can set an assertion that will fire whenever you have written or read a word in memory with incorrect parity. The syntax for this command is as follows.

```
mmassert_parity(m_id, "<access>", "<action>", <parity>, <start addr>, <end
addr>, <addr increment>);
```

Odd parity is encoded as “1” and even parity is encoded as “0” for the <parity> parameter.

Example: Print a message whenever I read from addresses 0 to 20 and the parity on the data is NOT odd (“1”).

Verilog

```
m_id = $mminstanceid("tb.mem.sdram0");
a_id = $mmassert_parity(m_id, "read", "note", 1, 0, 20, 1);
```

VHDL

```
m_id := mmgetinstanceid("/tb/mem/sdram0");
a_id := mmassert_parity(m_id, "read", "note", 1, 0, 20, 1);
```

Tcl

```
set m_id [mminstanceid tb.mem.sdram0]
mmassert_parity $m_id read note 1 0 20 1
```

5.3 Dynamically Enabling and Disabling Assertions

You can dynamically enable and disable (render inactive) assertions from within your test-bench using the following MMAV functions:

```
mmenableassertion(assertion_id);
mmdisableassertion(assertion_id);
```

The `assertion_id` is the variable that is assigned to the assertion definition call. In this example, the `assertion_id` would be “a_id”.

```
a_id = $mmassert_parity(m_id, "read", "note", 1, 0, 20, 1);
```

And to disable this assertion definition:

```
status = $mmdisableassertion(a_id);
```

Verilog

```
status = $mmenableassertion(a_id);  
status = $mmdisableassertion(a_id);
```

VHDL

```
status := mmenableassertion(a_id);  
status := mmdisableassertion(a_id);
```

Tcl

```
mmenableassertion a_id  
mmdisableassertion a_id
```

Usage:

For ease of adding assertions and modifying them for specific tests, Denali recommends that all assertions be placed in a single Verilog, VHDL or Tcl file and that all assertions be disabled using the “mmdisableassertion” call. Assertions can then be enabled individually in certain tests using the “mmenableassertion” call.

5.4 Error Injection Routines

You can insert errors into the Denali memories during simulation in order to exercise ECC checking logic and hardware test features in your design. There are two types of error injections routines. The first routine will randomly flip a random bit on the wires of a memory during a read at regular intervals. This routine simulates signal integrity errors in the real system. The second type of routines are hard fault routines that allow you to create stuck-at and coupling errors in the Denali memory models. This is useful when testing software or hardware that detects faulty memory devices in the real system. The syntax and description of these call are as follows:

5.4.1 Error Injection

Starting in Denali’s release 2.900, the error injection routine was enhanced to allow multi-bit errors and provide the user with greater flexibility over the randomization of errors.

NOTE: There is a limitation on the frequency of injected errors. Due to implementation, the smallest interval between injections is 2.

```
mmerrinject(<instance id>, "error_type_string");
```

“error_type_string” options:

-seed <seed_value>: Integer random number seed. By default, it uses system time as seed, thus generating a different result for each run. To get a repeatable sequence, this option should be used.

-reads <number_of_reads_per_error> <range>: Integer value for the number of reads per error. The optional second integer specifies the range, and the default value is 100.

-bits <bit number> <bit number 2> ...: The optional integer fields indicates different bit error numbers. For example, to create single and double bit errors, the values 1 and 2 would be specified. The default value is 1 (single bit errors only).

-percent <percentage integer> <percentage integer 2>: Each integer is the distribution/percentage of the occurrences of the bit errors as defined in **-bits** option above. The total sum of integers should be 100. By default, they are evenly distributed for different types.

Example: Generate single, double and 4-bit errors at a rate of 80% single bit errors, 15% double bit errors and 5% 4-bit errors. The errors will begin occurring randomly (using a seed of 12) 5-10 reads later.

Verilog

```
m_id = $mminstanceid("tb.mem.sdram0");
success = $mmerrinject(m_id, "-seed 12 -reads 5 10 -bits 1 2 4 -percent
80 15 5");
```

VHDL

```
m_id := mmgetinstanceid("/tb/mem/sdram0");
err_id := mmerrinject(m_id, "-seed 12 -reads 5 10 -bits 1 2 4 -percent 80
15 5");
```

Tcl

```
set m_id [mminstanceid tb.mem.sdram0]
mmerrinject $m_id -seed 12 -reads 5 10 -bits 1 2 4 -percent 80 15 5
```

Denali has also added a command to disable the error injection. To disable the error injection set using ***mmerrinject*** use the following command:

Verilog

```
success = $mmsetallerrinject(0);
```

VHDL

```
m_id := mmgetinstanceid("/tb/mem/sdram0");
```



```
success := mmseterrinject(m_id, 0); -- Turns OFF error injection
success := mmseterrinject(m_id, 1); -- Turns ON error injection
```

Tcl

```
set m_id [mminstanceid tb.mem.sdram0]
mmseterrinject $m_id 0
(OR)
mmsetallerrinject 0
```

For having error inject on every read, the command is:

```
$mmerrinject (id, 1); /* inject every read */
```

Similarly, if you need to inject a single bit error once every 512 reads then command is:

```
$mmerrinject (id, 512); /* inject error every 512th read*/
```

Enabling Error Injection on “backdoor” Reads

For backdoor reads (using mmreadword or from testbench tools such as Specman or VERA), the default behavior is to NOT do error injection. Error injection, by default, is only done on reads performed via the pins. However you can turn on error injection for backdoor reads by using a third, optional parameter of *mmseterrinject* in the following way:

```
mmseterrinject(id, status, [backdoorstatus]);
```

This function turns error injection on or off for the specified memory instance.

Arguments

- **id** - memory id for error injection, -1 for all
- **status** - 0 (to turn off) or 1 (to turn on)
- **backdoorstatus** - OPTIONAL - 0 (to turn off) or 1 (to turn on) default for backdoor reads is for error injection to be off (backdoor reads are those not through the pins but through the PLI or DDVAPI interface) error injection can be enabled on backdoor reads by setting the third parameter to 1.

Verilog

```
m_id = $mminstanceid("tb.mem.sdram0");
success = $mmsetallerrinject(m_id, 1, 1); // Turns ON backdoor error
injection
```

VHDL (currently only ModelSim VHDL is supported)

```
m_id := mmgetinstanceid("/tb/mem/sdram0");
success := mmseterrinject(m_id, 1, 1); -- Turns ON backdoor error
injection
```

Tcl

```
set m_id [mminstanceid tb.mem.sdram0]

mmtcleval("mmseterrinject $m_id 1 1");
```

5.4.2 Fault Modeling

Creating Memory Faults

Logical memory faults can be applied to any Denali memory device. These features can be very useful in testing fault logic and BIST (built-in self test) logic to verify memory arrays. The *mmfault* command is described below.

```
mmfault(<instance id>,"<type>",<addr>,<bit>,<value>,<slave addr>,<slave bit>);
```

The valid <type> fields are:

- stuck-at
A stuck at fault will fix the specified bit at the specified address to be stuck at the specified value.
- transition
A transition fault will insure that the specified bit at the specified address can never attain the specified value.
- coupling
A coupling fault will force a transition on the slave bit at the slave address whenever the specified bit on the specified address transitions.

Example: Set bit 5 at address 0x20 in memory instance tb.mem.sdram0 to stuck at 1.

Verilog

```
m_id = $mminstanceid("tb.mem.sdram0");

//Stick bit 5 of address=0x20 to "1"
err_id = $mmfault(m_id, "stuck-at", `h20, `h5, `h1);

//Make bit 1 of address=0x20 unable to transition to a '0'
err_id = $mmfault(m_id, "transition", `h20, `h1, `h0);

//In this example, a change in bit 0 at address 7 will cause a transition
//to that value at bit 2 at address 20 (if that bit does not already have
//the value).
err_id = $mmfault(m_id, "coupling", `h7, `h0, `h0, `h20, `h2);
```

VHDL

```
m_id := mminstanceid("/tb/mem/sdram0");

--Stick bit 5 of address=0x20 to "1"
err_id := mmfault(m_id, "stuck-at", 16#20#, 5, 1, 0, 0);

--Make bit 1 of address=0x20 unable to transition to a '0'
err_id := mmfault(m_id, "transition", 16#20#, 1, 0, 0, 0);

--In this example, a change in bit 0 at address 7 will cause a transition
--to that value at bit 2 at address 20 (if that bit does not already have
--the value)
err_id := mmfault(m_id, "coupling", 7, 0, 0, 16#20#, 2);
```

NOTE: *In VHDL, ALL fields must be used for any mmfault command. Thus unused fields must be set to "0".*

Tcl

```
--Stick bit 5 of address=0x20 (dec=32) to "1"
mmfault tb.mem.sdram0 stuck-at 32 5 1 0

--Make bit 1 of address=0x20 (dec=32) unable to transition to a '0'
mmfault tb.mem.sdram0 transition 32 1 0

# In this example, a change in bit 0 at address 7 will cause a transition
# to that value at bit 2 at address 20 (dec=32) (if that bit does not
# already have the value)
mmfault tb.mem.sdram0 coupling 7 0 0 32 2
```

Enabling/Disabling Fault Checks

Once enables, Denali memory faults are always active. The fault definitions can be disabled using the following commands.

mmsetfault(<fault id>, <flag>); - Turns on(flag=1)/off(flag=0) a specific fault specified by the fault id (err_id in the examples above)

mmsetallfault(<flag>); - Turns on(flag=1)/off(flag=0) all faults

Verilog

```
success = $mmsetfault (5, 0); //Turn off(0) fault id=5
success = $mmsetallfault(0); //Turn off all faults
```

VHDL

```
success = $mmsetfault (5, 0); //Turn off(0) fault id=5
success = $mmsetallfault(0); //Turn off all faults
```

Tcl

```
mmsetfault 5 0 #Turn off(0) fault id=5  
mmsetallfault 0 #Turn off all faults
```

5.5 Logical Addressing with MMAV (Method #1)

NOTE: *Logical memories, when created, generate a new instance name that is at the top of the testbench hierarchy. Thus when using other MMAV commands such as `mmwriteword`, `mmreadword`, `mmsave`, etc., you MUST use the instance name as specified in the logical memory definition, without any hierarchy. The “`address.space name=<name>`” XML tag identifies the instance “name” for the new logical memory instance.*

This section describes how to create system memories with MMAV. System memories can also be thought of as virtual, or logical, memories, but will be referred to as system memories throughout this document. They provide a virtual view of the physical memories in your design, to reorganize the data in a view that makes more sense for ease of debugging. For instance you can separate parity bits from data, change the bit order of addresses and data, interleave memories, and more.

In versions 3.0 and earlier, all system memories were created via PLI, VHDL or DDVAPI calls. There were multiple calls made to first create the memory, then add to it. This implementation had limitations, and required memories to be placed in an MxN grid, with no holes. The new specification is in very flexible XML. The 3.0 interfaces are still supported, but customers must use the new XML format if they wish to have more flexible organization of their memories. The 3.0 interface is described in Section 5.6, “Logical Addressing (Method #2),” on page 100.

5.5.1 XML Basics

XML (Extensible Markup Language) has a very simple syntax with a few basic rules. If you are familiar with HTML, it will look familiar. XML is not a language itself but a standard on how to define your own language, which Denali has done for creating system memories. The syntax is a combination of tags (delimited by `< >`), content, and attributes. The rules to remember when writing XML are:

- Every start-tag must have a matching end-tag or be a self closing tag
- Tags cannot overlap (they must be properly nested)
- XML documents can only have one root element
- XML is case sensitive

Denali can accept XML either from a file or from a string, see Section 5.5.8, “Interfacing to MMAV,” on page 100 for details.

Let's start with the root element. Your file or string containing XML must have one root element, and one Denali recognizes. Because Denali envisions supporting other things than system memories in XML eventually, the top level root is: "advanced.verification".

For example, the simplest XML for MMAV to process is:

```
<advanced.verification>
</advanced.verification>
```

which would do nothing.

Note that a matching end tag is required, to do this, begin the tag with / within the < >, with no space between the < and the /. Your XML must begin and end with this tag in order for MMAV to process it.

For defining system memories, the tag is: <system.addressing>. So, an empty system declaration would look like:

```
<advanced.verification>
  <system.addressing>
    <!-- This is a comment -->
  </system.addressing>
</advanced.verification>
```

Note that the indentation is not required but will be used in the examples for clarity.

Also, note that you can put comments within your XML. The syntax is:

```
<!--      comment      -->
```

5.5.2 Depth, Width Expansion

Now, within the system.addressing tags, you can create system memories.

Each system memory is defined with address.space tags. They must be given a unique name, a width, and a depth. Within the memory, you can place physical or other system memories.

Width Expansion example:

```
<advanced.verification>
  <system.addressing>
    <!-- Four 16mb X 8 bit memories, width expanded -->
    <address.space name='ram' width='32' depth='16777216'>
      <place baseaddr='0' bitpos='0'>
        <memory name='testbench.i0' />
      </place>
      <place baseaddr='0' bitpos='8'>
        <memory name='testbench.i1' />
      </place>
    </address.space>
  </system.addressing>
</advanced.verification>
```

```

    <place baseaddr='0' bitpos='16'>
      <memory name='testbench.i2' />
    </place>
    <place baseaddr='0' bitpos='24'>
      <memory name='testbench.i3' />
    </place>
  </address.space>
</system.addressing>
</advanced.verifikation>

```

This example shows how to create a system memory called "ram" 32-bits wide, 16Mb deep. This is created from 4 8-bit wide physical memories, placed side by side at bit positions 0, 8, 16, and 24.

Refer to the following figure that shows the XML Logical Memory Example.

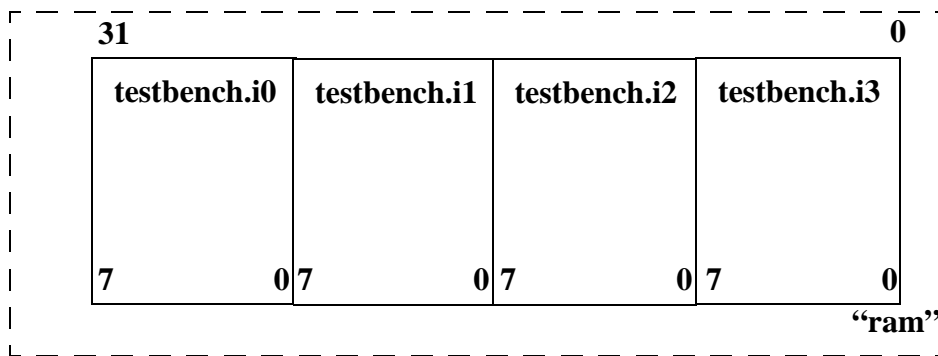


FIGURE 5-1: XML Logical Memory Example - Width Expansion

Depth Expansion Example:

```

<advanced.verifikation>
  <system.addressing>
    <!-- Two 32mb X 16 bit memories, depth expanded -->
    <address.space name='myview' width='16' depth='64M'>
      <place baseaddr='0' bitpos='0'>
        <memory name='testbench.i5' />
      </place>
      <place baseaddr='32M' bitpos='0'>
        <memory name='testbench.i6' />
      </place>
    </address.space>
  </system.addressing>
</advanced.verifikation>

```

which looks like:

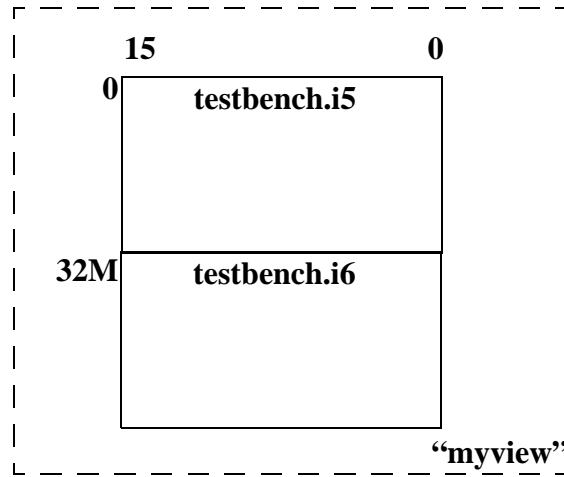


FIGURE 2. XML Logical Memory Example - Depth Expansion

5.5.3 Interleaving

Interleaving is accomplished by putting an `interleave` attribute on the place of affected memories, as well as adjusting the `baseaddr` accordingly.

Interleaving Example:

```
<advanced.verifcation>
  <system.addressing>
    <!-- Four 4k X 32 bit memories, interleaved -->
    <address.space name='cache' width='32' depth='16K'>
      <place baseaddr='0' bitpos='0' interleave='4'>
        <memory name='processor.c1' />
      </place>
      <place baseaddr='1' bitpos='0' interleave='4'>
        <memory name='processor.c2' />
      </place>
      <place baseaddr='2' bitpos='0' interleave='4'>
        <memory name='processor.c3' />
      </place>
      <place baseaddr='3' bitpos='0' interleave='4'>
        <memory name='processor.c4' />
      </place>
    </address.space>
  </system.addressing>
</advanced.verifcation>
```

which looks like:

	31	0
0	Processor.c1	
1	Processor.c2	
2	Processor.c3	
3	Processor.c4	
4	Processor.c1	
5	Processor.c2	
6	Processor.c3	
7	Processor.c4	
.....	
16380	Processor.c1	
16381	Processor.c2	
16382	Processor.c3	
16383	Processor.c4	

“cache”

FIGURE 3. XML Logical Memory Example - Interleave Expansion

5.5.4 Address Scrambling

You may also rearrange the bits of the address to suit your needs. Here is the example from Section 5.7, “Address Scrambling,” on page 105, reversing bits 4 through 7, and 12 through 15 of each address used to access tb.uut1 through scram1:

```
<advanced.verification>
  <system.addressing>
    <address.space name='scram1' width='24' depth='1M'>
      <place baseaddr='0' bitpos='0' >
        <address.map bits='19:16 12:15 11:8 4:7 3:0' />
        <memory name='tb.uut1' />
      </place>
    </address.space>
  </system.addressing>
</advanced.verification>
```

5.5.5 Data Bit Reordering and Masking

You may also view the data bits in a different order, and/or only use a subset of them in a given place.

```
<advanced.verification>
  <system.addressing>
```



```

<address.space name='data' width='30' depth='1M'>
  <place baseaddr='0' bitpos='0' >
    <data.map>
      <word bits='31:17 15:1' />
    </data.map>
    <memory name='tb.uut2' />
  </place>
</address.space>
<address.space name='parity' width='2' depth='1M'>
  <place baseaddr='0' bitpos='0' >
    <data.map>
      <word bits='16 0' />
    </data.map>
    <memory name='tb.uut2' />
  </place>
</address.space>
</system.addressing>
</advanced.verification>

```

Note that you could instead combine these together to make a 32-bit memory, with parity on the right by doing the following and changing the width on the address.space to 32:

```

<word bits='31:17 15:1 16 0' />

```

5.5.6 Creating Holes

This specification also allows for leaving 'holes' in the system memory - spaces that are not occupied by any physical or child system memory. The space does not need to be fully specified. For instance, to leave the bottom 64K of a system memory empty, simply start placing memories at a baseAddr equal to 64K.

5.5.7 Putting it all Together

Here's an example tying these concepts together:

A 256kB x 8 memory, addressed by big-endian 32-bit words, width-expanded with a 32kB x 64, addressed by 32-bit words, in little-endian order:

```

<advanced.verification>
  <system.addressing>
    <address.space name='wide' width='64' depth='64K'>
      <place baseaddr='0' bitpos='0' >
        <data.map>
          <word bits='7:0 15:8 23:16 31:24' />
        </data.map>
        <memory name='narrow' />
      </place>
    </address.space>
  </system.addressing>
</advanced.verification>

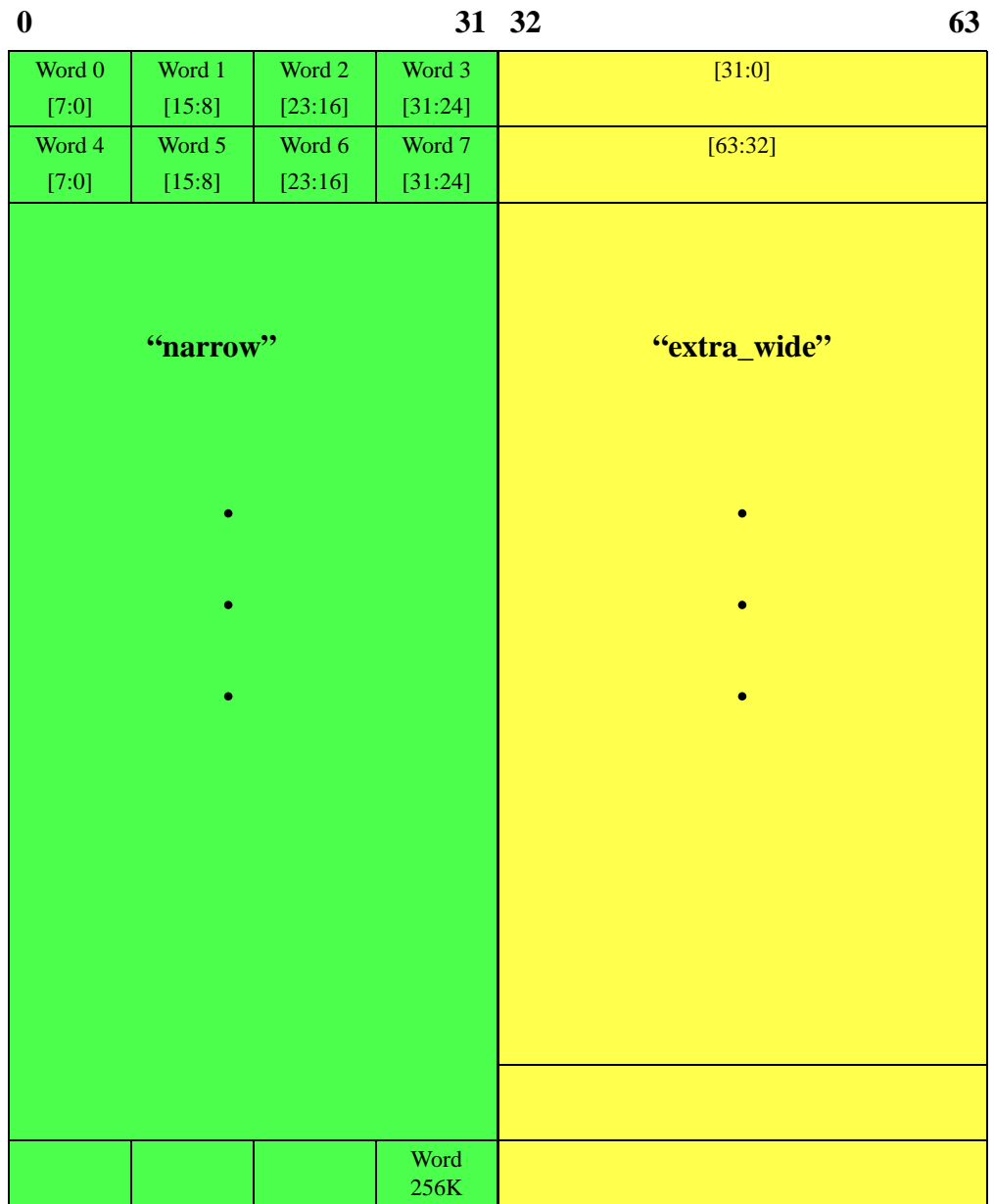
```

```

    <data.map>
      <word bits='31:0' />
      <word bits='63:32' />
    </data.map>
    <memory name= 'extra_wide' />
  </place>
</address.space>
</system.addressing>
</advanced.verification>

```

Logical Memory Diagram



The examples should make much of the specification clear, but let's look at each XML element in detail. Each element is followed by a description of its attributes.

<system.addressing> A container for logical addressing descriptions.

<address.space> Describes a logical address space. (mandatory: name, width, depth)

- name - Identifier for the address space akin to instance names for physical memories.
- width - Number of bits in each word in this address space.
- depth - Number of words in the address space.

<memory> Refers to a physical memory instance. (mandatory: name)

- name - Instance name of the physical or system child memory.

<place> Specifies placement attributes of a physical memory or logical address space within a logical address space. (mandatory: baseaddr; optional: bitpos, interleave)

- baseaddr - Address in the logical space at which the child begins.
- bitpos - Bit number (LSB = 0) in the logical space corresponding to the child's LSB. Defaults to 0.
- interleave - Numeric offset in the logical space between consecutive locations in the child. For example, with an interleave of two, the child's words will correspond to every alternate logical address. Interleave another child with this one by also setting the second's interleave='2' and its baseaddr one greater than the first's baseaddr. Defaults to 1; that is, no addresses are skipped.

<data.map> Modify the enclosing placement (i.e., <place>) to allow viewing the child other than by full words in their natural bit order (width-1 : 0).

<word> - One or more <word>s in a <data.map> describe the bit arrangement for viewing the placed memory. (mandatory: bits)

- bits - Space-separated list of bit numbers and bit ranges. Bits are numbered starting at zero for the least-significant bit. A bit range is two bit numbers joined by a colon. The range represents all the bits numbered between the two numbers, inclusive, and the range is in either ascending or descending order, depending on whether the left number is less than or greater than that, respectively. (For completeness, the bit range N:N should probably be considered equivalent to the simple bit number N.)

With a single <word>, one may rearrange the bits in the placed memory into a different logical order. To view the memory through a wider bus width than its natural width, the “bits” attribute may include bit numbers beyond (width-1). A bit number $N * \text{width} + B$ indicates bit B in the Nth successive word. See the first <word> in the example above.

To view the memory through a narrower-than-native bus width, use multiple **<word>**s, using the “bits” attributes to indicate how to apportion the bits of a word among the logical words. See the second **<data.map>** in the example above.

<address.map> Modify the enclosing placement to allowing viewing the child other than in consecutive order of ascending addresses. (mandatory: bits)

bits - Space-separated list of bit numbers and bit ranges. Bits are numbered starting at zero for the least-significant bit. See the bits attribute of **<word>**, above, for a discussion of ranges.

To understand how the **<address.map>** is applied, consider the computation of an address in the placed memory without any address mapping. From an address in the enclosing address space, subtract the child's baseaddr, divide by its interleave, and multiply by the ratio of child words to logical words as determined by any **<data.map>** in the placement. This gives the address in the child memory to which a logical address corresponds, or, in the case of a **<data.map>** that combines bits from multiple child locations, the first such address. When an **<address.map>** is applied to the placement, it is this computed address whose bits are rearranged as specified by the bits attribute.

5.5.8 Interfacing to MMAV

Now that you have these memories defined, either in a file, or in a string in your testbench, you can create them in the Denali environment in the following ways:

Verilog:

```
success = $mmxmleval("xmlString");  
success = $mmxmlfile("filename");
```

VHDL:

```
success := mmXmLEval(xmlString);  
success := mmXmLFile(filename);
```

You can make multiple calls to the routines, all your system memories do not have to be in the same file or string buffer, though they can be.

You can find out what the instance id of this memory is using **\$mminstanceid**, for use in subsequent calls which take the id number versus the name.

5.6 Logical Addressing (Method #2)

Prior to Section 5.5, “Logical Addressing with MMAV (Method #1)” in Denali MMAV version 3.1, you could define simple width expanded, depth expanded, and interleaved memories using testbench PLI/FLI or Tcl calls. The first command is used to specify the

"shape" of the logical address. Succeeding commands are then used to map specific physical memories to the logically addressed memory.

Once logical addressing has been created, the standard PLI/FLI/Tel functions for reading, writing, saving and restoring can be used on the logically addressed memory in the same way as with physical memories. The exception is with the compare command.

The logical addressing supported by these commands are created by combinations of:

width expansion

Two or more physical memories are combined to form a logical address word with more bits. The physical memories combined in this way need not be of the same width, but must have equal address spaces.

depth expansion

Two or more physical memories are combined to form a larger address space for the logically addressed memory (equivalent to the sum of all the address spaces for the constituent memories). The physical memories combined in this way need not have equal address spaces, but must have equal word size.

interleaving

Two or more physical memories may be interleaved so that stepping through the addresses in the logically addressed memory will result in cycling over the interleaved physical memories. The physical memories that are interleaved together must have equal width and depth.

masking

Masking may be used to select only some of the bits in the word of a physical memory in defining the logical addressing word. These bits need not be contiguous in the memory, but there is no mechanism for reordering the bits. The masking is fixed for each physical memory that comprises the logically addressed memory. Also, if there is depth expansion, the masks must be consistent over all the physical memories involved in the depth expansion. An example of when masking might be used is when the physical memory word represents two (or more) different values. For example, the first 4 bits could represent a priority value, or a pointer, and the remaining bits could represent data. Using the appropriate masks, two logically addressed memories could be defined corresponding to those two values. The resulting logically addressed memories are then, in general, MxN arrays of physical memories, with or without interleaving, and with or without masking.

```
id = $mmcreatesystemmem( "logically addressed memory name",  
                           "logically addressed memory instance id",  
                           width,  
                           depth,  
                           [numinterleaves] );
```

- **logically addressed memory name** - user specified name for the logically addressed memory instance name
- **logically addressed memory instance id** - user specified instance id for the logically addressed memory. **NOTE: This is the instance id that will be used with subsequent MMAV commands, such as mmload, mmsave, mmwriteword, mmreadword, etc.**
- **width** - number of physical memories used in width expansion
- **depth** - number of physical memories used in depth expansion
- **interleaves** - the level of interleaving

This command is used to create the configuration of the logically addressed memory. The physical memories are laid out in an “MxN” array, with or without interleaving. An id is returned that corresponds to the logically addressed memory. Note that the logically addressed memory is not ready for use after this command, since no physical memories have yet been mapped to the logically addressed memory. The following commands are used for that purpose.

```
result = $mmaddtosystem( <logically addressed memory id>,
                        <instance id>,
                        <width position>,
                        <depth position>,
                        [<interleave position>])
```

This command is used to add a physical memory to a logically addressed memory previously created by a mmcreatesystem command. A command of this kind must be issued for every position in the matrix as defined by the mmcreatesystem command before the memory is usable.

Arguments:

logically addressed memory id - the id of the logically addressed memory as returned by mmcreatesystem

instance id - the id of the physical memory being added into the logically addressed memory

width position - the width position in the MxN matrix where this memory should go (values should be 0 to width-1)

depth position - the depth position in the MxN matrix where this memory should go (values should be 0 to depth-1)

interleave position - the level of interleave for this memory (values should be 0 to interleave-1)

This function returns a 0 on success, and a -1 on failure.

```

result = $mmaddtosystemmask( <logically_addressed_memory_id>,
                             <physical_id>,
                             "mask string" ,
                             <width position>,
                             <depth position>,
                             [interleave position])

```

This function is identical to `mmaddtosystemmem` (described above) except that a string is specified that represents the binary representation of the mask. Recall that the mask must be the same for all physical memories of a given width position.

This function returns a 0 on success, and a -1 on failure.

Examples:

Verilog

```

m_id1 = $mminstanceid("tb.mem.sdram0");
m_id2 = $mminstanceid("tb.mem.sdram1");
m_id3 = $mminstanceid("tb.mem.sdram2");
m_id4 = $mminstanceid("tb.mem.sdram3");

vid = $mmcreatesystem("logical", "new_memory_name", 2, 2, 1);
success = $mmaddtosystemmem(vid, m_id1, 0, 0, 0);
success = $mmaddtosystemmem(vid, m_id2, 1, 0, 0);
success = $mmaddtosystemmem(vid, m_id3, 0, 1, 0);
success = $mmaddtosystemmem(vid, m_id4, 1, 1, 0);

```

VHDL

```

m_id1 := mmgetinstanceid("/tb/mem/sdram0");
m_id2 := mmgetinstanceid("/tb/mem/sdram1");
m_id3 := mmgetinstanceid("/tb/mem/sdram2");
m_id4 := mmgetinstanceid("/tb/mem/sdram3");

vid := mmcreatesystem("logical", "new_memory_name", 2, 2, 1);
success := mmaddtosystemmem(vid, m_id1, 0, 0, 0);
success := mmaddtosystemmem(vid, m_id2, 1, 0, 0);
success := mmaddtosystemmem(vid, m_id3, 0, 1, 0);
success := mmaddtosystemmem(vid, m_id4, 1, 1, 0);

```

Tcl

```

set m_id1 [mminstanceid tb.mem.sdram0]
set m_id2 [mminstanceid tb.mem.sdram1]
set m_id3 [mminstanceid tb.mem.sdram2]
set m_id4 [mminstanceid tb.mem.sdram3]

set vid [mmcreatesystem logical new_memory_name 2 2 1]
mmaddtosystemmem $vid $m_id1 0 0 0
mmaddtosystemmem $vid $m_id2 1 0 0

```

```
mmaddtosystemmem $vid $m_id3 0 1 0
mmaddtosystemmem $vid $m_id4 1 1 0
```

In the above examples, a logically addressed memory is created from a 2x2 array of physical memories.

In the following example, two logically addressed memories (data and pointer) are created from two 8 bit wide physical memories using masks. The first logically addressed memory consists of the 5 most significant bits from the first physical memory. In the example, this represents a 5 bit pointer. The second logically addressed memory is created from the least significant bits of the first physical memory and all the bits of the second physical memory. In the example, this represents an 11 bit logically addressed word.

Examples:

Verilog

```
//Pointer
m_id1 = $mminstanceid("tb.mem.sdram0");
m_id2 = $mminstanceid("tb.mem.sdram1");

vid = $mmcreatesystemmem("logical", "data", 1, 1, 1);
success = $mmaddtosystemmemmask(vid, m_id1, "11111000", 0, 0, 0);

//Data
vid1 = $mmcreatesystemmem("logical", "pointer", 2, 1, 1);
success = $mmaddtosystemmemmask(vid1, m_id1, "00000111", 0, 0, 0);
success = $mmaddtosystemmemmask(vid1, m_id2, "11111111", 1, 0, 0);
```

VHDL

```
-- Pointer
m_id1 := mmgetinstanceid("/tb/mem/sdram0");
m_id2 := mmgetinstanceid("/tb/mem/sdram1");

vid := mmcreatesystemmem("logical", "data", 1, 1, 1);
success := mmaddtosystemmemmask(vid, m_id1, "11111000", 0, 0, 0);

-- Data
vid1 := mmcreatesystemmem("logical", "pointer", 2, 1, 1);
success := mmaddtosystemmemmask(vid1, m_id1, "00000111", 0, 0, 0);
success := mmaddtosystemmemmask(vid1, m_id2, "11111111", 1, 0, 0);
```

Tcl

Pointer

```
set m_id1 [mminstanceid tb.mem.sdram0]
```



```

set m_id2 [mminstanceid tb.mem.sdram1]

set vid [mmcreatesystemem logical data 1 1 1]
mmaddtosystemmask $vid $m_id1 11111000 0 0 0

-- Data

set vid1 [mmcreatesystemem logical pointer 2 1 1]
mmaddtosystemmask $vid1 $m_id1 00000111 0 0 0
mmaddtosystemmask $vid1 $m_id2 11111111 1 0 0

```

5.7 Address Scrambling

Address Scrambling is used to create a “scrambled” memory from a physical memory. The scrambling consists of a bit-by-bit translation of the original address to the scrambled address. This allows the user to re-define the way that a particular address gets mapped into memory.

Example:

```

mid = $mmaddressmap("scrambled memory name", memory_id, "<address map>",
[output message level]);

```

Where:

- **scrambled memory name** = Newly created scrambled memory for the scrambled addresses
- **memory_id** = Physical memory that is to be mapped to the “scrambled” memory
- **“address map”** = Bit-by-bit translation from the physical to the “scrambled” address
- **[output message level]** = [Optional] Verbose messages showing the physical to “scrambled” addresses translation

The first integer in the address map specifies the value of the most significant bit of the scrambled address. In other words, the first integer specifies which bit of the original address is used for the most significant bit of the scrambled address. Similarly, the last integer specifies the least significant bit. For example, to reverse the bits in a 4 bit address, the address map would be: “0 1 2 3”. Here, the MSB of the scrambled address is given by bit 0 of the original address. The LSB is given by bit 3 of the original address. For example, with this mapping the address 0xa would scramble to 0x5 (1010 -> 0101).

Valid integers in the address map are in the range 0 .. n, where n is the number of bits required to represent the largest addressable memory location in the memory instance. For example, if 0x1ffff is the largest addressable memory location, you would need to specify 21 mapping integers. In this case, valid values for the mapping integers are from 0 to 20.

To display verbose messages when scrambling, use a value of 1 for the **[output message level]** argument in \$mmaddressmap. This causes messages like the following to appear in standard output:

```
*Denali* (write) scramble: address 255 maps to 8160
```

By default, no such messages appear in stdout.

Verilog

```
mid = $mmaddressmap( "scram1", id1, "19 18 17 16 12 13 14 15 11 10 9 8 4  
5 6 7 3 2 1 0", 1 );
```

This command will scrambled all writes and reads to the scrambled memory "scram1", which consists of the physical memory 'id1'. The scramble scheme consists of reversing bits 4 through 7, and 12 through 15 of a given address.

VHDL

```
mid := mmaddressmap( "scram1", id1, "19 18 17 16 12 13 14 15 11 10 9 8 4  
5 6 7 3 2 1 0", 1 );
```

Tcl

```
set id1 [mminstanceid tb.mem.sdram0]  
mmaddressmap scram1 $id1 "19 18 17 16 12 13 14 15 11 10 9 8 4 5 6 7 3 2 1  
0" 1
```

5.8 Scratchpad Memories

Scratchpad memories are used to create a simple C-based MxN memory array. This can be extremely useful in modeling caches, embedded memory arrays, etc.

```
$mmcreatescratchpad(<name>, <width>, <depth>, <init_val>);
```

- **name** - name of the scratchpad returned by *mmcreatescratchpad*
- **width** - width of the scratchpad in bits (integer)
- **depth** - number of words in the scratchpad (integer)
- **init_val** - initial memory value of scratchpad (0, 1, x, X, "")

This command is available through the PLI or via Tcl. The init_val argument is not optional, though it can be the empty string "", in which case MMAV uses a default value of "X".

Example: Create a 32-bitx1024 scratchpad memory called "scratchpad1".

Verilog:

```
sp_id = $mmcreatescratchpad (scratchpad1, 32, 1024, 1);
```

VHDL:

For VHDL, Denali has created two direct functions in *func_if.vhd* (for NC-VHDL) and in *func_mti.vhd* (for MTI)

```
FUNCTION mmCreateScratchpad( name : STRING;
                             width : INTEGER;
                             depth : INTEGER;
                             initvalue : STD_LOGIC_VECTOR ) return INTEGER;
attribute FOREIGN of mmCreateScratchpad:function is "Clib:mmcreatescratchpad";
```

If you put this call within an entity/architecture that gets instantiated more than once, you can make the path names to the newly created scratchpad memories unique by using the VHDL 'path_name attribute.

Example:

```
entity simple_mem is
port (
    a      : in std_logic_vector (9 downto 0);
    data   : inout std_logic_vector(7 downto 0);
    cs, we, clk : in std_logic
);
end simple_mem;

use work.memory_modeler.all;

architecture behavior of simple_mem is
    signal width: integer := 8;
    signal size:integer := 1024;
    signal id: integer := -1 ;
begin

    process
    begin
        id <= mmcreatescratchpad(simple_mem'path_name & "storage", width,
size, "00000000");
        wait;
    end process;
```

When this is instantiated in the testbench twice like this:

```
i_simple_mem: simple_mem
port map( a => tb_a, data => tb_data1, cs => tb_cs, we => tb_we, clk
=> tb_clk);

j_simple_mem: simple_mem
port map( a => tb_a, data => tb_data2, cs => tb_cs, we => tb_we, clk
=> tb_clk);
```

You will get unique path names which to refer to the names as follows:

```
*Denali* Class: scratchpad Instance:
":simple_tb:j_simple_mem:simple_memstorage" Size: 1Kx8

*Denali* Memory id: 0 created of size 1024, width 8.
*Denali* Class: scratchpad Instance:
":simple_tb:i_simple_mem:simple_memstorage" Size: 1Kx8

*Denali* Memory id: 1 created of size 1024, width 8.
```

Alternatively, you can refer to the memories by the id returned from `mmcreatescratchpad`.

Tcl:

```
set sp_id [mmcreatescratchpad scratchpad1 32 1024 1]
```

The above example creates a scratchpad memory named `scratchpad1` that is 32-bits wide and 1024 words deep (each of which contains FFFFFFFF until changed). The variable `sp_id` contains the integer value returned by the command that is the id of the memory created. The following will appear in the Denali history file:

```
*Denali* Memory id: 1 created of size 1024, width 32.
```

You can then use **`mmreadword`** and **`mmwriteword`** (and other MMAV calls) referring to "1" (not recommended) or `sp_id` (`$sp_id` if Tcl) to access this memory from your testbench. In addition, PureView can be also be used to view this memories contents.

5.9 Verilog Callbacks (new in 3.2)

5.9.1 Callback Interface

The MMAV Verilog callback interface provides the facilities for model callback initialization and handling. The **`denaliMemCallback`** module contains an integer variable which will be changed by the model when a sensitized event occurs in the model. A change on this variable should trigger a procedural block in the testbench to process the callback events.

5.9.2 Callback Initialization

To use the Denali MMAV Verilog callbacks, you must compile the *`$DENALI/ddvapi/verilog/ddvapi.v`* file along with your other Verilog files. The tasks included in this file provide the necessary tasks to implement the callback routines.

Callbacks are generated whenever an MMAV assertion is triggered. When setting MMAV assertions, you must use the “**callback**” <action> type. See Figure 5.1, “Setting Assertions on Memory Transactions,” on page 82 for details on how to setup callback assertions.

To set up a callback on a model event (or a set of model events), instantiate a **denaliMemCallback** module to be attached to the callbacks for the model instance. There should be separate callback instances corresponding to each model instance so that callbacks for each model instance may be processed by independent procedural blocks.

With the **denaliMemCallback** module is instantiated, a model callback is set up by calling **setCallback()** with the model instance ID to attach the enabled events to the **denaliMemCallback.Event** variable.

Example (setup a callback whenever a read or write occurs to instance testbench.uut1):

```
denaliMemCallback cb    ();
denaliMemTrans    trans ();
integer id0, asrt0, i;

initial
begin
    id0 = $mminstanceid ("testbench.uut1");
    asrt0 = $mmassert_access (id0, "ReadorWrite", "callback", 0,
'h7FFFFFFF);

// Enable All the Callback Types
    for (i = 0; i < DENALI_CB_TOTAL; i = i + 1)
        cb.enableReason [i] = i;

    cb.setCallback (id0);

    #10000000000;
    $finish;
end
```

5.9.3 Callback Handling

Processing of callbacks from the model is performed in a procedural always-block triggered by the Event variable in the **denaliMemCallback** instance. At any one time in the simulation, multiple callback events may have occurred simultaneously at different points in the model. The callback model handles the list of events triggered in the model, populating the callback module register variables on each successive **getCallback()** call. This function returns 1 as long an additional event exists, and returns zero when the last event has been processed.

On each call of the **getCallback()** function, the reason and transaction ID (**transId**) variables are loaded with the next available callback event. The **transId** can then be passed to

the **denaliMemCallback transGet()** task to retrieve the actual packet associated with the event.

```
always @(cb.Event)
begin
    while (cb.getCallback(id0))
    begin
        trans.transGet (cb.transId);

        $display ( "time      : %0t" , $time );
        $display ( "reason    : %0s" , cb.reasonStr (cb.reason) );
        $display ( "assertion : %0h" , trans.assertion );
        $display ( "reason    : %0h" , trans.reason );
        $display ( "address   : %0h" , trans.address );
        $display ( "width     : %0h" , trans.width );
        $display ( "data      : %0h" , trans.data );
        $display ( "mask      : %0h" , trans.mask );

        $display;
    end

    $display;
end
```

5.9.4 denaliMemCallback Registers

enableReason

enableReason is an array of 32-bit values to specify all of the callback events to be enabled. These values must be loaded by the user prior to calling the **setCallback()** task. **setCallback()** will step through this array starting with location **enableReason[0]**, and will enable each callback event specified up to the first uninitialized location. Any **enableReason[]** values above the first uninitialized location will be ignored.

The values loaded in to this array must be taken from the set of available callback events defined in the **DENALIDDVCBpointT** section of **ddvapi.vh**, (i.e. **DENALI_CB_***).

Event

Event is an integer variable which will be attached to the model callback function. Any time one of the enabled internal events occurs in the model, the Event variable will change. A value change on this variable should trigger a procedural block in the testbench to process the current callback event(s).

reason

reason is an integer variable identifying the current callback event; this variable is loaded by the **getCallback()** function. When multiple callback events have occurred at the same simulation time, reason will be updated to identify the next available event on each subsequent **getCallback()** call.

transId

transId is an integer variable identifying the transaction associated with the current callback event; this variable is loaded by the **getCallback()** function. When multiple callback events have occurred at the same simulation time, **transId** will be updated to identify the transaction for the next available event on each subsequent **getCallback()** call.

5.9.5 denaliMemCallback Tasks and Functions

setCallback()

Setup a callback in a model instance, attaching the selected callback events to the **Event** variable in the **denaliMemCallback** module. Prior to calling **setCallback()**, the desired callback events to be enabled must be initialized in the **enableReason** array.

```
task setCallback;  
input [31:0] instId;
```

instId is the Denali instance ID (handle), identifying a particular MMAV instance.

getCallback()

Step through the callback events from the model for the current simulation time. Set the reason and **transId** variables to identify the next available event on each successive **getCallback()** call. The return value is 1'b1 if an event is found, or 1'b0 if all events have been processed and there are no more events available.

```
function getCallback;  
input [31:0] instId;
```

instId is the Denali instance ID (handle), identifying a particular MMAV instance.

clearCallback()

Clears the given callbacks for an Denali instance ID.

```
function clearCallback;  
input [31:0] instId;
```

instId is the Denali instance ID (handle), identifying a particular MMAV instance.

reasonStr()

Return the callback reason as a printable ASCII string.

```
function [30*8:1] reasonStr;
  input [31:0] reason;
  case (reason)
    DENALI_CB_None           : reasonStr = "None";
    DENALI_CB_Read           : reasonStr = "Read";
    DENALI_CB_Write          : reasonStr = "Write";
    DENALI_CB_ReadOrWrite    : reasonStr = "ReadOrWrite";
    DENALI_CB_ReadNoWrite    : reasonStr = "ReadNoWrite";
    DENALI_CB_ReadRead       : reasonStr = "ReadRead";
    DENALI_CB_WriteWrite     : reasonStr = "WriteWrite";

    default                   : reasonStr = "INVALID";
  endcase
endfunction
```

reason must be one of the **DENALIDDVCBpointT** values defined in \$DENALI/ddvapi/verilog/ddvapi.vh. For example:

```
$display ( "access : %0s" ,  cb.reasonStr (cb.reason));
```

will display one of the “reason” strings listed above.

5.10 Using MMAV with Mentor Graphic's Seamless HW/SW Co-Verification Product

Mentor Graphic's Seamless Hardware/Software Co-Verification product comes bundled with a version of Denali's MMAV. In the past, to use a new Denali release with Seamless, customers had to wait for a new Seamless release. With the new releases of both Denali and Seamless, this is no longer the case.

To use a new Denali release with your current Seamless release, simply install the new Denali release in **\$CVE_HOME/denali**. Alternatively, you can also symbolically link **\$CVE_HOME/denali** to a Denali MMAV release elsewhere. This will allow you to change releases easier and keep old ones around.

To use Seamless and Denali's PureView together, refer to Chapter 2: "Using the PureView Graphical Tool" and Chapter 3: "Debugging Memory Using PureView".

5.11 Using MMAV for Embedded ASIC Memories

This section describes how to create embedded ASIC memories with MMAV.

Denali's embedded memory solutions provide vast improvements in the usability and verification features over typical vendor models. Denali provides simulation models via its MMAV product for register files, embedded SRAM, embedded DRAM, and embedded Flash. These high quality models provide the same features as Denali's de-facto external memory models.

The primary advantage of using Denali's embedded memory solutions is that these models work with all Denali verification and debug tools.

Some of the key features of all Denali memory models:

- Preload the memories from a file
- Do “back-door” reads and writes to the models
- Compare the memory contents to a “golden” file
- Save off the memory contents to a file
- Setup assertions on memory transactions
- Setup error injection and BIST features
- Use PureView to interactively view and edit the memory contents

One other key advantage to Denali's models is the small simulation footprint. Typically vendor models use a large static array for the memory storage whereas Denali uses dynamic memory allocation and only allocates the memory being used during the simulation.

You can use Denali's PureView graphical debugger to view your embedded memory contents during simulation and edit the contents “on-the-fly” during simulation.

5.11.1 Register Files

Using PureView, you can easily create your own register files. Refer to the figure below:

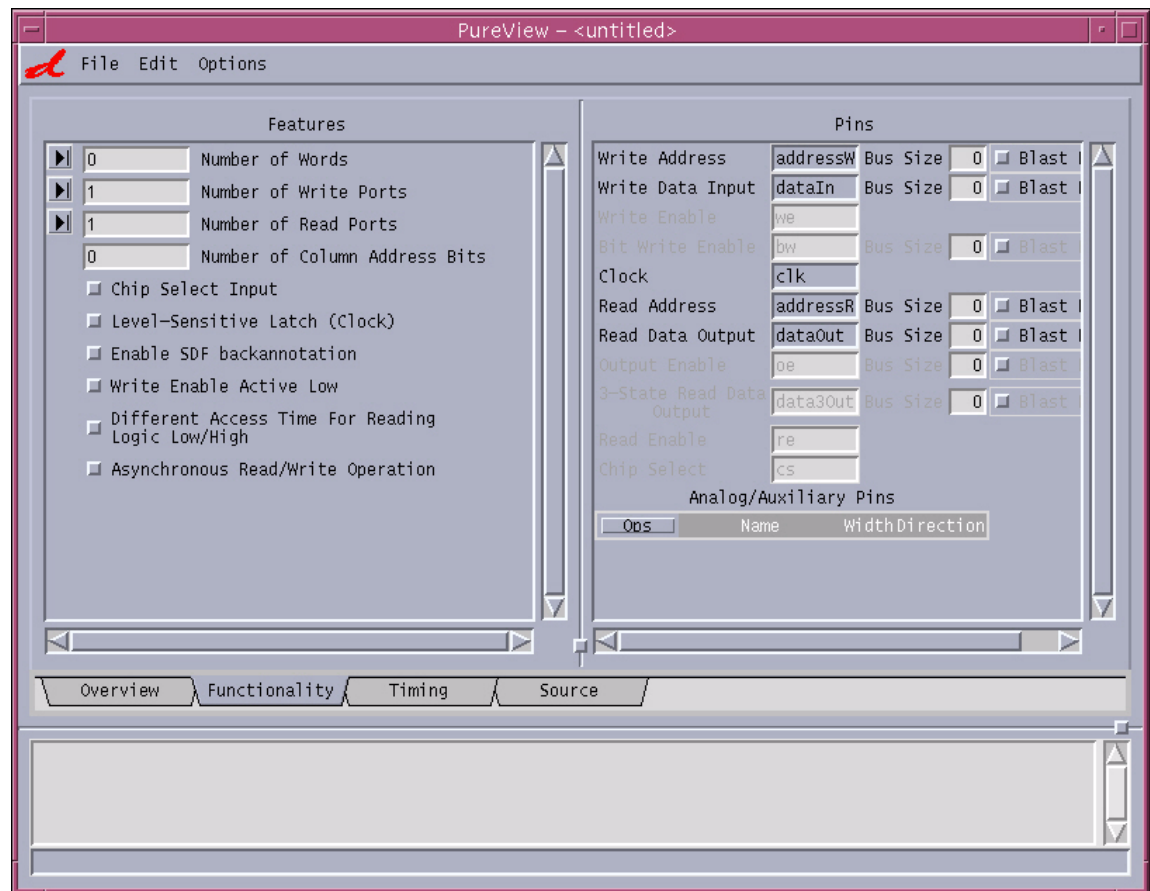


FIGURE 5-2: PureView Register File Section GUI

Notice that there is also an **Auxiliary** pin section that enables you to define other test pins, etc. that are common on embedded devices. This allows you to create an exact Verilog or VHDL shell to hookup to your design. By simply comparing the features of the register file from the ASIC suppliers datasheet, you can easily create a Denali register file model to instantiate in your design. Once you have added a Denali register file into your design, you can now take advantage of all the memory model features that have made Denali the de-facto memory model solution.

5.11.2 Embedded SRAM

Denali's Embedded SRAM (essram) model is a generic model which intends to cover as many libraries as possible. Usually, a vendor's memory has a subset of features of the model described here.

The following is a screen shot of the embedded SRAM model from PureView.

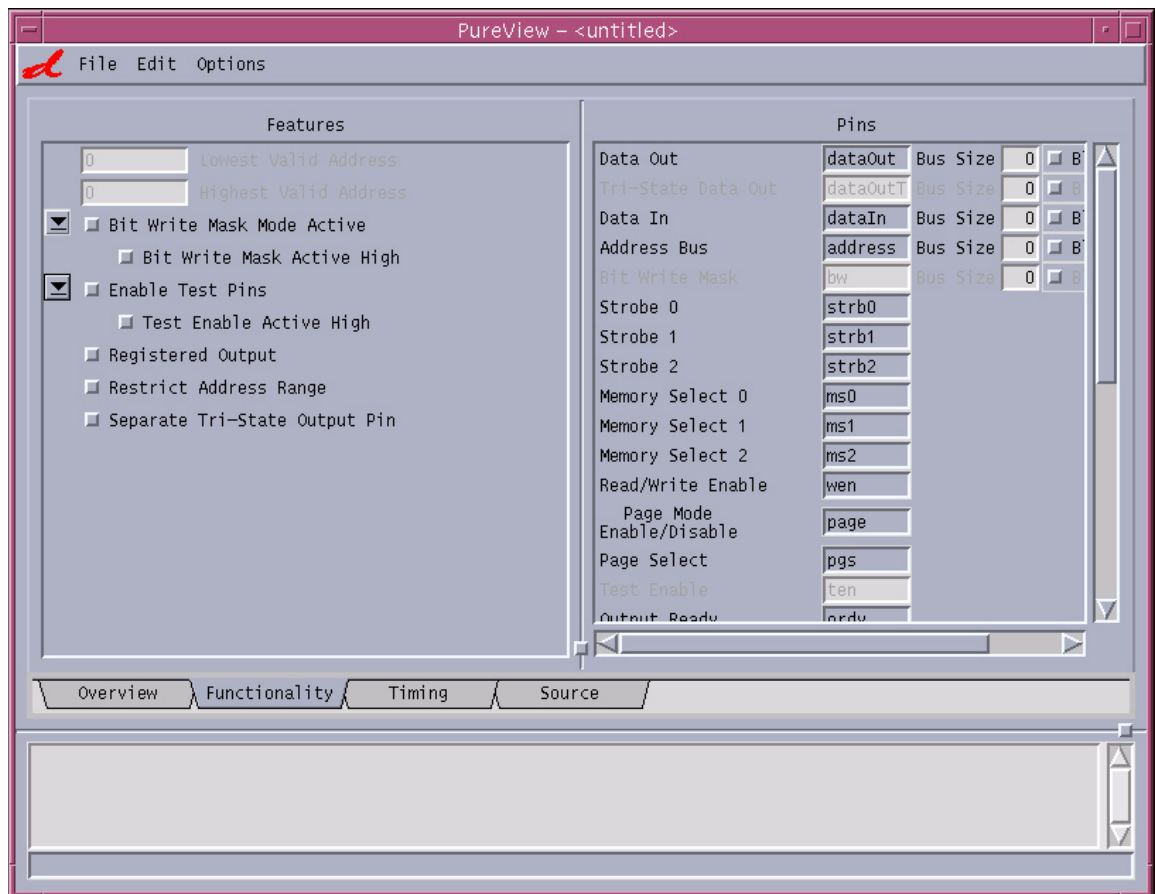


FIGURE 5-3: PureView Embedded SRAM GUI

Denali's embedded SRAM model enables you to define multiple data ports and provides the flexibility to configure these as read, write or read and write.

The fields **Read Pins Bit String** and **Write Pins Bit String** should be specified for I/O port bit map. "1" means port available, "0" means port unavailable. For example, for a single port memory part, you can specify:

```
"Read Pins Bit String" = "1"
"Write Pins Bit String" = "1"
```

This indicates that it has one data-in pin and one data-out pin. For a dual-port memory, if you specify:

```
"Read Pins Bit String" = "10"
"Write Pins Bit String" = "01"
```

This indicates that port0 has one data-in pin and no data-out pin and port1 has no data-in pin and one data-out pin.

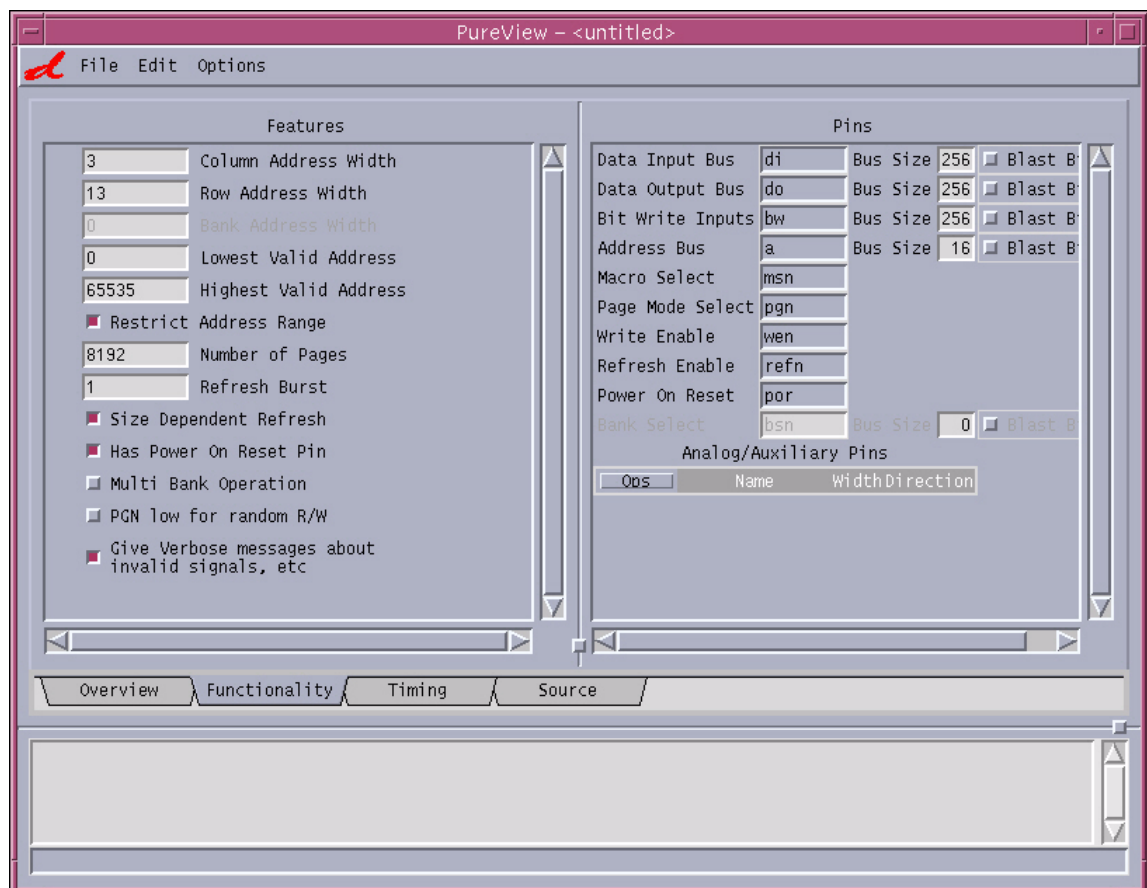
For a dual-port memory, if you specify:

```
"Read Pins Bit String" = "11"  
"Write Pins Bit String" = "11"
```

This indicates that port0 has one data-in pin and one data-out pin (read/write) and port1 has one data-in pin and one data-out pin.

5.11.3 Embedded DRAM

Denali also provides an embedded DRAM model. At this time, this model fully covers IBM's SA27-E and Cu-11 embedded DRAM model. Below is the PureView screen shot for the IBM embedded DRAM model.



PureView Embedded DRAM GUI

6 MMAV Testbench Integration

This chapter describes MMAV's integration with various supported testbench interfaces.

6.1 Verilog Interface

MMAV provides a direct Verilog integration using the PLI. Using PureView, you can generate a Verilog wrapper for your memory model instances. For details, refer to [“Using the PureView Graphical Tool” on page 11](#).

NOTE: *Currently, Denali only supports the wrappers being of all in one language or another. For example, in a mixed-mode simulation, if you have a Verilog wrapper, you can instantiate it in a Verilog testbench only. In case you have a VHDL wrapper, you can instantiate it in the VHDL testbench only.*

MMAV also provides a way to generate callbacks to your Verilog testbench on memory accesses. Refer to [“Simulation Environment” on page 21](#) for more details.

6.1.1 Simulating with MMAV and Verilog

Cadence Verilog-XL

To link Cadence Verilog simulator with the models, run the `vconfig` script provided with the Cadence release. This script will prompt you for several options, you will simply answer the questions appropriate to set-up your environment. If you are unsure about an answer, simply hit <RETURN> which will automatically select the default value.

When the script asks for your `veriusers.c` file, enter: `${DENALI}/verilog/veriusers.c`, for example:

```
The user template file 'veriusers.c' must always be included in the link
statement. What is the path name of this file?
[veriusers.c] : ${DENALI}/verilog/veriusers.c
```

When asked for other files to be linked into the Verilog executable, enter: `${DENALI}/verilog/denverlib.o`, for example:

```
List the files one at a time, terminating the list with a single '.'
-----> ${DENALI}/verilog/denverlib.o
-----> .
```

After completing the program, you will have a script named *cr_vlog*. Run this script to create a new Verilog executable, which includes the Denali models.

Refer to an example compile script shown below:

```
#!/bin/csh -f
#
# Script to create : Dynamic PLI library
#
set verbose
mkdir $DENALI/verilog/lib

# Verilog XL
gcc -c $DENALI/verilog/veriusers.c \
    -I$CDS_INST_DIR/tools/verilog/include \
    -o $DENALI/verilog/veriusers.o

# Linking step

ld -G \
    $DENALI/verilog/veriusers.o \
    $DENALI/verilog/denverlib.o \
    -o $DENALI/verilog/lib/libpli.so

unset verbose
```

Cadence NC-Verilog

Statically linking Denali PLI with NC-Verilog (for all platforms):

1. Set the CDS_INST_DIR environment variable to your NC-Verilog installation directory.
% setenv CDS_INST_DIR <nc_install_dir>
setenv INSTALL_DIR \$CDS_INST_DIR
setenv ARCH to sun4v or lnx86 as appropriate
2. Copy the file *\$CDS_INST_DIR/tools/inca/files/Makefile.nc* to a *Makefile* in your working area:
% cp \$CDS_INST_DIR/tools/inca/files/Makefile.nc Makefile
3. Edit this copy of the *Makefile* as follows:
Edit the VERIUSER_C statement to point to Denali's copy:
VERIUSER_C = \$(DENALI)/verilog/veriusers.c
Add the Denali PLI object modules in the statement:
PLI_OBJECTS = \$(DENALI)/verilog/denverlib.o

Comment out the ARCH_RELOCATE_OPT definition that is used for dynamic linking:

```
# ARCH_RELOCATE_OPT = ...
```

4. Build the static executables:

```
% make static
```

Synopsys VCS

To run MMAV with VCS, you need to link in the Denali memory models with the simulator. Unlike other Verilog PLI's, VCS does not use a **veriusers.c**. Instead, the simulator learns about what PLI functions exist by reading a PLI table.

To run VCS, you need to only add to your existing command line where to find the **pli.tab** file, and the object files for MMAV to be linked in, as shown in the example below.

```
vcs -M -P $DENALI/verilog/pli.tab \  
-LDFLAGS "$DENALI/verilog/denverlib.o" \  
<your verilog files>
```

NOTE: *Certain platforms (notably Linux) also require linking with the "dynamic linking" library (using -ldl) as below:*

```
vcs -M -P $DENALI/verilog/pli.tab \  
-LDFLAGS "$DENALI/verilog/denverlib.o" -ldl \  
<your verilog files>
```

You may then simulate using *simv*.

Denali VCS PLI table files

In `$DENALI/verilog`, there are 2 VCS PLI table files, *pli.tab* and *pli-fast.tab*.

The *pli.tab* file uses the "*" wildcard to indicate that the scope of where ACC capabilities are enabled is for the entire design.

In general, if you are not having initialization performance issues, continue to use `$DENALI/verilog/pli.tab`.

The *pli-fast.tab* file uses the %TASK wildcard to limit the scope of where ACC capabilities are enabled, thereby speeding up the initialization of the simulation run.

If you are using this file and get an error message like:

```
Error: acc__handle_by_name: Object memory_spec not found
```

then you are using one of the Denali tasks in more than one scope and should enable the appropriate module or by using the wildcard "*" as in `$DENALI/verilog/pli.tab`.

Mentor Graphics ModelSim

Specifying PLI Applications for Modelsim

Since PLI applications are dynamically loaded by the simulator, you must specify which applications to load (each application must be a dynamically loadable library). For details, refer to *Compiling and linking PLI/VPI C applications* in the *ModelSim User's Manual*.

NOTE: *The pliapp2 and pliappn references are NOT Denali PLI applications, but the examples are shown to illustrate how to link multiple PLI objects with Denali's PLI.*

There are three ways in which you can specify PLI applications for ModelSim. The Denali PLI shared object file for ModelSim is **mtipli.so** for Unix/Linux. Usage for these shared libraries are specified as follows:

1. As a list in the **Veriuser** entry in the *modelsim.ini* file:

Unix:

```
Veriuser = $DENALI/mtipli.so pliapp2.so pliappn.so
```

2. As a list in the **PLIOBJS** environment variable:

Unix:

```
setenv PLIOBJS "$DENALI/mtipli.so \  
pliapp2.so pliappn.so"
```

3. As a -pli argument to the simulator (multiple arguments are allowed):

Unix:

```
% vsim -pli $DENALI/mtipli.so -pli pliapp2.so -pli pliappn.so
```

The various methods for specifying PLI applications can be used simultaneously. The system libraries are loaded in the order listed above. Environment variable references can be used in the paths to the libraries in all cases.

ModelSim Timescale

When running simulations with ModelSim, in order to have the Denali timestamp recorded correctly, the ModelSim simulator uses the minimum time precision used in the design modules.

This means that you should not specify a time precision using the “-t” timescale command-line option. Using the “-t” timescale option will result in erroneous timing error messages as the timestamp that Denali uses for timing checks will be incorrect.

Aldec Riviera-PRO and Active-HDL

You can instantiate the Denali models as Verilog modules. The models communicate with Riviera-PRO via the **denali.so** shared object library (Solaris, Linux). The library is delivered with both Riviera-PRO and Active-HDL.

To instantiate the Denali memory models in Verilog designs:

- Set `$DENALI` environment variable to point to the Denali installation directory. You must set this variable prior to running the simulator (Riviera-PRO or Active-HDL).
- Add the **denali.so** shared object library (Solaris, Linux) to the list of PLI applications visible to the simulator.
- Denali Verilog procedures and tasks for controlling memory models are located in the **denali.so** shared object library (Solaris, Linux). If you use these tasks and functions in your Verilog code, make sure that they receive appropriate arguments.

*In case Denali memory model exits abruptly when it detects a fatal error (e.g., when a file with the model configuration cannot be found), it may terminate the simulation and close the simulator. If you run simulation at the command-line mode, a message explaining the cause of the error appears in the OS console window. If you use Riviera-PRO GUI or Active-HDL, then the GUI closes. To find out the cause of the error, refer to the **denali.error** file that is created in the current directory.*

6.2 VHDL Interface

MMAV provides a direct VHDL integration through each simulator specific C interface. Using PureView, you must generate a VHDL wrapper for your memory model instances by choosing the simulator you are using, as the wrappers have information specific to each simulator. For details, refer to [“Using the PureView Graphical Tool” on page 11](#).

NOTE: *Currently, Denali only supports the wrappers being of all in one language or another. For example, in a mixed-mode simulation, if you have a Verilog wrapper, you can instantiate it in a Verilog testbench only. In case you have a VHDL wrapper, you can instantiate it in the VHDL testbench only.*

6.2.1 Simulating with MMAV and VHDL

Synopsys VCS

In order to access Denali calls from within the VHDL, you must:

1. Compile the file `$DENALI/vhpi/memory_modeler.vhd`
2. Include the following in the top level of your entity:

```
use work.Memory_Modeler.all;
```

Mentor Graphics ModelSim

To run simulation with MMAV and VHDL using Mentor Graphics ModelSim, do the following:

```
#!/bin/csh -fx
rm -rf work
vlib work
vcom $DENALI/mti/memory_modeler.vhd
vcom *.vhd
vsim -c testbench -do 'run -all'
```

Cadence NC-VHDL (Leapfrog)

To run simulation with MMAV and VHDL using Cadence NC-VHDL, do the following:

```
/bin/rm -rf WORK
mkdir WORK
ncvhdl -v93 -m -work WORK $DENALI/leapfrog/memory_modeler.vhd
ncvhdl -v93 -m -work WORK *.vhd
ncelab -messages -update WORK.TESTBENCH:BEHAVIOR
ncsim -messages -input -logfile simulation.log -LOADFMI $DENALI/
libdenfmi:den_FMIptr WORK.TESTBENCH:BEHAVIOR
```

Aldec Riviera-PRO and Active-HDL

You can instantiate the Denali models as VHDL modules.

NOTE: *The previous versions of Riviera-PRO, up to version 2007.06 and Active-HDL, up to version 7.3 supported Verilog instantiation only.*

The models communicate with Riviera-PRO via the **denali.so** shared object library (Solaris, Linux). The library is delivered with both Riviera-PRO and Active-HDL.

To instantiate Denali memory models in VHDL designs:

- Set \$DENALI environment to point to the Denali installation directory. You must set this variable prior to running the simulator (Riviera-PRO or Active-HDL).
- In the VHDL wrapper file generated by PureView (for details, refer to [“Using the PureView Graphical Tool” on page 11](#)), edit the foreign attribute of the architecture so that it has the following form:

```
attribute foreign of <architecture> : architecture is "VHPI \
denali;initDenali"
```

PureView does not currently list Riviera-PRO or Active-HDL under **Options > Simulation Environment > VHDL**. You can generate a wrapper file for any other simulator available on the list and edit it so that the foreign attribute matches the specification shown above.

The following example shows a sample model generated by PureView that is then edited to match Aldec's requirements (file *regfile001.vhdl*).

```
-- Entity: regfile001
-- SOMA file: /home/joe/mems/regfile001.soma
-- Initial contents file:
-- Simulation control flags:
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;
ENTITY regfile001 IS
    GENERIC (
        memory_spec: string := "regfile001.soma";
        init_file:   string := "init.txt";
        sim_control: string := ""
    );
    PORT (
        addressWR : in STD_LOGIC_VECTOR(14 downto 0);
        dataIn    : in STD_LOGIC_VECTOR(7  downto 0);
        we        : in STD_LOGIC;
        clk       : in STD_LOGIC;
        addressRD : in STD_LOGIC_VECTOR(14 downto 0);
        dataOut   : out STD_LOGIC_VECTOR(7  downto 0);
        cs        : in STD_LOGIC
    );
END regfile001;
ARCHITECTURE behavior of regfile001 is
    attribute foreign: string;
    attribute foreign of behavior: architecture is "VHPI denali;
    initDenali";
BEGIN
END behavior;
```

- Denali VHDL procedures and functions for controlling memory models are located in the **memory_modeler.vhd** package. If you use these functions in your VHDL code, compile package **Memory_Modeler** and add appropriate **library** and **use** statements. The source code for the package is available in the **memory_modeler.vhd** file located in the Riviera-PRO or Active-HDL installation directory.

NOTE: *Contact Aldec for more information on this interface.*

6.3 SystemC Interface

MMAV provides a direct SystemC integration. Using PureView, you can now generate a SystemC wrapper for your memory model instances. Denali's SystemC Integration uses its Yukon C-interface for vastly improved simulation performance.

6.3.1 MMAV and SystemC Overview

SystemC is a C++ simulation environment primarily used for system level design. It is a modeling platform consisting of C++ class libraries and a simulation kernel for design at behavioral and register-transfer levels. For more information on SystemC itself, refer to <http://www.systemc.org>.

MMAV integration with SystemC environment consists of the following:

- A SystemC wrapper for the MMAV model
- Denali library
- An integration file that is used to interface between the wrapper and the Denali Yukon library.

You can generate the SystemC wrapper for the MMAV model by using the PureView interface. Select **Options > Simulation Environment > SystemC**. Once the wrapper is generated, you can instantiate the top-level `sc_module` in a SystemC testbench.

6.3.2 Simulating with MMAV and SystemC

Refer to `$DENALI/yukon/example` directory for an example using the DDR class of memory models. It also contains a sample testbench and a **Makefile** to link MMAV and SystemC together.

The *yukon* directory in the *\$DENALI* release contains the relevant Denali libraries (*libyukon.o* and *libyukon.so*).

Denali recommends that you compile and run the example as described in the *README* file in that directory. This will ensure your environment is properly set up and you have an appropriate Denali license working. You must be using the GNU C++ compiler version 2.95 or later. The SystemC example has been tested with version 2.95.2.

Once you have successfully run the example, copy the example over and make the required modifications to fit your environment and usage.

There are three C++ files included in the example:

- *main.cc*

This is an example of a testbench. This would be replaced by your actual testbench. As shown in the example, it is important to call `DENALITerminate` at the end of the simulation in order to dump all pending transactions correctly...

- *simulator.cc*

This is the file which interfaces Denali to SystemC via our Yukon interface.

- *ddr.cc*

This is a Denali wrapper for the model generated using PureView.

6.4 Specman Interface

This section provides details on how to use MMAV within a Specman environment.

To use the callback methods described in this document, you must be running MMAV version 3.2 with Specman 3.3.1 or higher.

To get more detailed information on using Denali MMAV with Specman, refer to *Specman Elite Denali MMAV Interface Guide*; Chapter 14 of the *Usage and Concepts Guide for Specman Elite*. You can access these guides from `docs` directory of Cadence Specman release.

6.4.1 MMAV and Specman Overview

You can access or capture activity to and from the Denali memory models.

In the Specman Elite/ Denali Interface (SNDI), there are two different ways of communicating with a Denali memory from your e testbench using:

- Data-Driven Verification (DDV) methods that have been ported to e
- Callbacks

You can access memory using Denali DDV API functions.

To access a Denali Memory Model from Specman, you must first create an HDL shell using Memory Maker as normal and instantiate this shell in your testbench or DUT. For details, refer to [“Using the PureView Graphical Tool” on page 11](#).

There are two predefined, user-accessible e structures for the SNDI interface, named `sn_denali` and `sn_denali_unit`. There is one global instance of `sn_denali` that has 4 methods associated with it to interact with the memory models on a global basis. The second struct, `sn_denali_unit` is an abstract unit type that requires you to inherit from in order to gain access to the built-in read and write methods. The way to do this is with the Specman Elite `SN_DENALI_MEMORY_UNIT` template macro. The macro requires you to specify the widths. The following example shows how to do this:

```
<`
SN_DENALI_MEMORY_UNIT demo_denali64 using width = 64; // model width must
                                                         // be defined
extend sys {

mem1: demo_denali64 is instance;
    keep mem1.hdl_path() == "/top/mem1";
};
`>
```


SNDI is automatically initialized when the first instance of `sn_denali_unit` is generated. The initialization loads the Denali shared library, initializes the Denali simulator, and binds the `SN_DENALI_MEMORY_UNIT` instances to their API counterparts.

The following figure shows the basic organization of Specman, Denali and your verification environment.

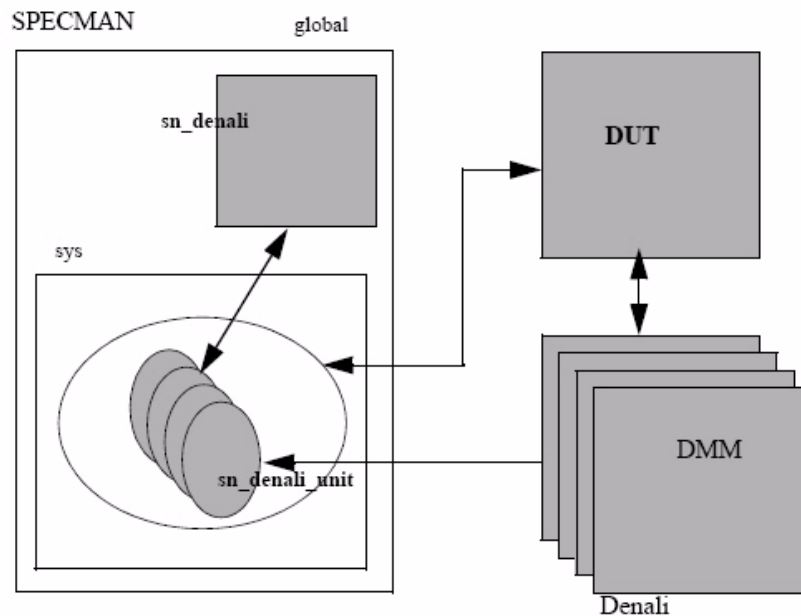


FIGURE 6-1: Verification Environment

6.4.2 Simulating with MMAV and Specman

Mentor Graphics ModelSim

To run simulation with MMAV and Specman using Mentor Graphics ModelSim, do the following:

1. Setup up the environment variables

```
SPECMAN_AUTO_PRE_COMMANDS=false; export SPECMAN_AUTO_PRE_COMMANDS
SN_AUTO_PRE_COMMANDS=false; export SN_AUTO_PRE_COMMANDS
```
2. Create Shared lib `libqvl_sn_basic.so` including Denali

```
sn_compile.sh -sim qvl -l "$DENALI/mtipli.so"
```
3. Use shared lib created above

```
SPECMAN_LIBQVL=`pwd`/libqvl_sn_basic.so; export SPECMAN_LIBQVL
```
4. Clean up before compiling

```
rm -rf verilog/specman.v work
vlib work
```
5. Create Verilog stubs file

```
specman -commands "define VERILOG_ENV; load e_code/mem_top.e; write
stubs -verilog verilog/specman.v;"
```

6. Compile verilog

```
vlog verilog/tb.v verilog/essram.v verilog/specman.v +incdir+verilog
```

7. When the following command launches Specview and ModelSim GUIs, do the following:

- Type `sn` then return at the simulator GUI prompt.
 - In the Specman GUI load a testcase (*\$DENALI/example/specman/e_code/mem_tc1.e* or *e_code/mem_tc2.e*).
 - Then click Test in the Specman GUI.
 - Switch to the simulator GUI by hitting Return.
 - Type `run -all` at the ModelSim GUI prompt to run the simulation
- ```
specview -p "define VERILOG_ENV;\
set wave -mode=interactive mti; load e_code/mem_mti_wave; \
load e_code/mem_top;" \
vsim -keepstdout tb specman specman_wave &
```

## Synopsys VCS

To run simulation with MMAV and Specman using Synopsys VCS, do the following:

1. Create stubs file `specman.v`

```
specman -commands "define VERILOG_ENV; load e_code/mem_top.e; write
stubs -verilog verilog/specman.v;"
```

2. Compile simulator-specman linked executable including Denali:

```
sn_compile.sh -sim vcs \
-vcs_flags "+incdir+verilog" \
-vcs_flags "-I verilog/tb.v verilog/essram.v verilog/specman.v" \
-vcs_flags "-P $DENALI/verilog/pli.tab" \
-vcs_flags "-LDFLAGS $DENALI/verilog/denverlib.o"
```

3. When the following command launches Specview and VCS GUIs:

- type `$sn` then return at the simulator GUI prompt.
  - In the Specman GUI load a testcase (*e\_code/mem\_tc1.e* or *e\_code/mem\_tc2.e*). Refer to *\$DENALI/example/specman*.
  - Click Test in the Specman GUI.
  - Switch to the simulator GUI by hitting Return.
  - Add HDL signals to the waveform viewer.
  - Type `.` at the vcs GUI prompt to run the simulation:
- ```
specview -p "define VERILOG_ENV; load e_code/mem_top.e;" vcs \
-Mupdate -o ./vcs_specman -RIG verilog/tb.v verilog/essram.v \
verilog/specman.v &
```

6.4.3 Configuration Register and Memory Access

The Specman integration with MMAV interface enables you read from or write to memory.

SNDI Methods

Specman Elite/Denali Interface methods are the MMAV functions that have been ported to the `sn_denali` and `sn_denali_unit` types through the DDVAPI. This integration provides access to Denali Data-Driven Verification interface without having to leave the Specman environment. It is important to point out that not all MMAV functions have been integrated into Specman. Only Specman Denali predefined methods are available within e code.

It is also important to point out that since e does not understand unknown's, the SNDI interface has developed methods to read and write unknown values into the Denali memory model. For example, `read_unk()`, `write_unk()`.

For details on `sn_denali` and `sn_denali_unit` methods, refer to *using_sn_denali_interface.pdf* in your Specman installation directory.

6.4.4 Extending `sn_denali_unit` to include all MMAV Functions as Methods

You will notice that not all functions for MMAV have been ported over to methods of the `sn_denali_unit`. It is easy to extend the `sn_denali_unit` to include these methods. A simple code snippet demonstrating this is below.

```

// Sample Code Snippet for extending sn_denali_unit

<`
define SN_DENALI_ADDR_SIZE    32; //Define the Max Addr Size in Bits
define SN_DENALI_DATA_SIZE    512; // Define the Max Data Size in Bits
define SN_DENALI_SMALL_STRING_SIZE 96; // Define the Max String Size in
Bits
define SN_DENALI_STRING_SIZE 2048; // Define the Max String Size in Bits

extend sn_denali_add_on {

    instance : string;
    mem_id : uint;
    !verilog_instance : list of byte;

    str_e2v(e_string : string): list of byte is {
        var packed_str: list of byte;

        packed_str = pack(packing.network, e_string);

        result = packed_str[1..]; // skip the first 0 byte
    }; // End Method str_e2v
}; // End Struct sn_denali_add_on

extend sn_denali_unit {

run() is also {
    add_on.instance = me.full_hdl_path();
    add_on.verilog_instance = add_on.str_e2v(add_on.instance);
    add_on.mem_id = me.get_id();

};

verilog function '~/$mmassert_uma' ( action :
    SN_DENALI_SMALL_STRING_SIZE ) : 1;

assert_uma ( action : string ) @sys.any is {
    var packed_val : list of byte;
    var status    : uint (bits:1);
    packed_val = add_on.str_e2v(action);
    status = '~/$mmassert_uma' (packed_val) ;

    check that (status == 0) else dut_error(appendf("%12d ns (%s) ERROR:
        Denali method call to assert_uma(%s) failed RC=%d",
        sys.time,add_on.instance,packed_val,status));

};

`>

```

The above e file does not extend all the functions of MMAV as methods to `sn_denali_unit`, but does give a representative set. With the addition of the MMAV functions, you can build a test environment around your memory subsystem.

The methods defined above can be found in `$DENALI/example/sndi_example/sn33_denali.e`.

6.4.5 Viewing Memory Transactions in Waveforms

One big advantage of using Specman Elite is that in addition to viewing signals in a waveform window, you can also view events or transactions on the memory. The following figure shows that actions for `memory_0` are displayed in the waveform window.

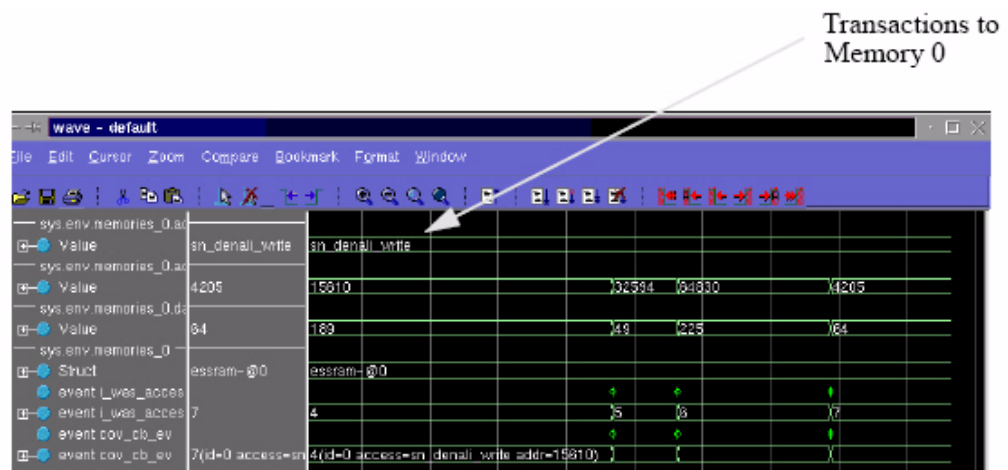


FIGURE 6-2: Waveform Showing Events in Waveform Window

6.4.6 Example Testcase

You can access some basic example with callbacks from `$DENALI/example/specman` directory. These include examples for SRAM, SDRAM, SMROM, and ESSRAM.

You can also find a detailed example of SNDI at `$DENALI/examples/specman`. This testcase utilizes the system memories from Denali along with the callbacks from SNDI.

6.5 Vera Interface

This section provides details on how to use MMAV from a Vera testbench.

To get more detailed information on using Denali MMAV with Vera, refer to *\$VERA_HOME/lib/denali/README_DDVAPI*.

6.5.1 MMAV and Vera Overview

Denali's Data-Driven Verification (DDV) methods have been ported to Vera. For details on MMAV-Vera functions and tasks, refer to *\$VERA_HOME/lib/denali/denali_ddv.vrh*.

Some of the functions available in MMAV are not integrated into the MMAV-Vera interface. You can access these Denali functions by calling these through the Denali TCL interpreter. The Vera DirectC function `DenaliDDVTclEval()` can execute a TCL routine through the Denali TCL interpreter. This TCL routine can call any one of the Denali functions or execute any TCL script.

The following example shows a TCL script with this call.

```
// Tcl call from within Vera Testbench
integer iErr;
iErr = DenaliDDVTclEval ("source MyDenaliTclFile.tcl");
/ Tcl Routine -- MyDenaliTclFile.tcl
mmsaverange /tb/mem/sdram0 save.dat 0 31
mmcomp /tb/mem/sdram0 golden.dat
```

The above example will save the contents of addresses 0 to 31 of memory instance */tb/mem/sdram0* to file *save.dat* and then compare the contents of the entire memory */tb/mem/sdram0* to a golden file *golden.dat*.

Architecture

The following figure illustrates the MMAV-Vera interface architecture.

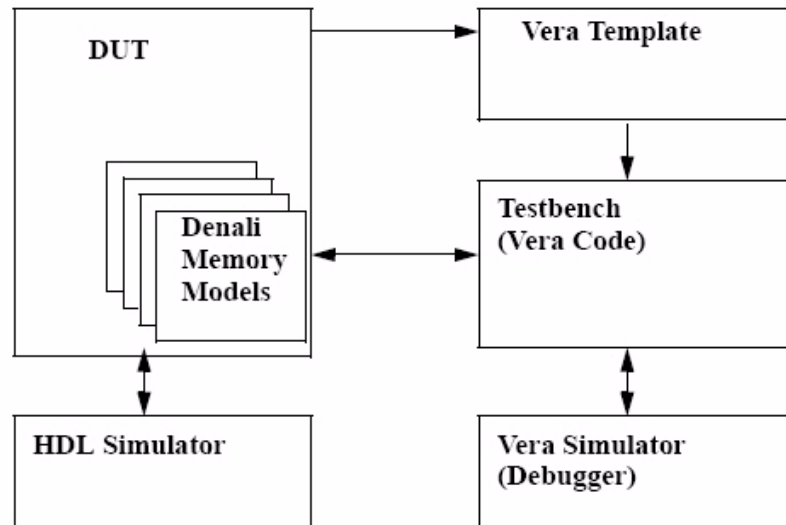


FIGURE 6-3: MMAV and Vera Architecture

6.5.2 Simulating with MMAV and Vera

In order to use MMAV and Vera with your simulator, you must first link in the appropriate libraries according to the instructions in the manuals.

The steps required to use MMAV within your Vera environment are as follows:

1. To access a Denali Memory Model from Vera, you must first create an HDL shell using PureView and instantiate this shell in your testbench or DUT. For details, refer to [“Using the PureView Graphical Tool” on page 11](#).
2. In order to gain access to the DDV functions that has been ported to Vera, you must include the header file in your Vera testbench.

```
#include <denali_ddv.vrh>
```

NOTE: MMAV no longer supports the old CAPI interface. The MMAV Vera interface uses DDVAPI. For details on DDVAPI functions, refer to `$DENALI/doc/ddvapi.pdf`. This manual also includes details on how to transition from CAPI to DDVAPI. The Vera document `$VERA_HOME/lib/denali/README_DDVAPI` covers the changes from CAPI to DDVAPI.

3. Initialize Denali memory from Vera testbench. For details, refer to [“Initializing Denali Memory Models from Vera Testbench” on page 136](#).
4. Setup callbacks (optional). For details, refer to [“Processing Callbacks” on page 137](#).

5. Call other Denali functions from Vera testbench, as needed (optional). For details refer to [“Using other Denali functions from Vera Testbench” on page 139](#).
6. Link Denali, Vera, and simulator together and run your testbench.

Synopsys VCS

To run simulation with MMAV and Vera using Synopsys VCS, do the following:

```
#!/bin/sh -f
vera -cmp display.vr
make -f Makefile_vcs
```

NOTE: *The Makefile_vcs is available in \$DENALI/example/vera/denali_vera.tar.gz. The \$VERA_HOME/lib/denali/README_DDVAPI file also has run scripts for different platforms.*

6.5.3 Initializing Denali Memory Models from Vera Testbench

To access the Denali Memory Models with any of the functions, you must first initialize the simulator API from within Vera. This must be done in order for the Denali Memory Models to talk to Vera and must be called prior to issuing any other Denali functions, such as pre-loading memory, issuing backdoor reads or writes, dumping the contents of memory, etc.

Initialization of the Memory Simulator API needs to be called only once in the simulation. You can do this by simply calling the `DENALDDVInitialize` function from within Vera.

```
DENALIDDVinitialize (string init, string reportFunc, var integer
unknownsP, string elabDone, string clientName);
```

Argument	Type	Description
init	string	These two arguments specifies the strings which name the exported VERA tasks to be registered as Denali initialize and access callbacks. A null-string argument disables that callback.
reportFunc	string	
unknownsP	var integer	<p>This variable tells the simulator how to store unknowns in the address content data. It can take the form of three options.</p> <ul style="list-style-type: none"> unknownsNBit1 - unknown bit will be stored per bit of data unknowns1Bit1 - unknown bit will be stored per address content of data unknowns0Bit0 - unknown bit will be stored per address content of data

Argument	Type	Description
elabDone	string	Specifies the name of the exported VERA task to be called by Denali when simulation is about to start. This is only set on the rare occasions when the client plans on taking ownership of any memories. In general, it should not be set. A null-string argument disables this callback.
clientName	string	Specifies the client name.

For example:

```
integer iErr;
iErr = DenaliDDVinitialize( "DramInitCbK", "DramAccCbK1", iUnkMode, "",
"DDVAPI Denali Vera Example");
```

In this example, iErr returns 0 on success.

6.5.4 Processing Callbacks

Callback tasks can either be declared during Initialization or later on with the `AccessSet-Callback` function call.

There are 3 types of callbacks in the Vera and MMAV interface as following:

- Initialization callback
- Access callback
- Iterate callback

Refer to `$VERA_HOME/lib/denali/README_DDVAPI` for details.

Here is a callback example:

```
integer iErr;
iErr = DenaliDDVaccessSetCallback("regCbFunc");

export task regCbFunc(integer id, integer access, integer portNum)
{
    bit [31:0] bData;
    bit [31:0] bMask;
    bit [63:0] address;
    integer iWidth, iErr;
    integer memId;

    iErr = DenaliDDVgetIdByName("testbench.i0", memId);

    if (id == memId) {
        iErr = DenaliDDVaccCbkGetDataAndMask( iWidth, bData, bMask,
address);
    }
}
```

In this example, iErr returns 0 on success.

You can disable all the callbacks or just a particular memory callback.

To disable all callbacks:

```
iErr = DenaliDDVTclEval("mmdisablecallbackall");
```

To disable callback for a particular memory:

```
iErr = DenaliDDVTclEval("mmdisablecallback -user <instance_id>");
```

In case you disable all callbacks or only one particular memory instance, you can enable callbacks for each one of them:

```
iErr = DenaliDDVTclEval("mmenablecallback -user <instance_id>");
```

Blocking Assignments in Callbacks

The Vera exported task that you set as a callback function is expected to return without blocking. Such blocking can occur explicitly, such as with "@(posedge clock)", or implic-

itly, such as by reading or writing an HDL signal, which causes blocking to the clock edge associated with the signal.

If you need to run blocking code in response to a callback, that code can be spawned from the callback task in a separate thread with the fork-join-none construct. However, the main callback thread must still return without blocking.

You can write your callback code so that it blocks. In such a situation, a second callback may occur while the first callback is still active. Vera detects this situation and discards the second callback.

Vera detects the blocked-callback error at two times: first when the callback fails to return "asynchronously" (without blocking), and again if a second callback occurs while the first callback is still active. In both cases, Vera's only indication that an error has occurred is to return `DENALIError_Unknown` instead of `DENALIError_NoError` to Denali.

However, if you have made Vera DirectC function call `DenaliDDVwarnDiscarded-Cbk(1)`; then an error or warning message will be printed in either case.

6.5.5 Using other Denali functions from Vera Testbench

You can use other Denali functions from Vera testbench, as needed. This is optional.

Refer to the example below. In this, `DenaliDDVgetIdByName` returns the instance id corresponding to a particular instance name and `DenaliDDVload` loads that memory instance with data from the file *file.dat*. It also shows how to do `mmsomaset` using `DenaliDDVTclEval`.

```
integer id;
integer iErr;
integer unk = 0;

string inst = "testbench.i0";
string file = "file.dat";
string init = "";
string access = "";

iErr = DenaliDDVinitialize("", "", unk, "", "API");
iErr = DenaliDDVgetIdByName(inst, id);
iErr = DenaliDDVload(id, file);
iErr = DenaliDDVTclEval("mmsomaset testbench.i0 tds 0.4 ns");
```

In this example, `iErr` returns 0 on success.

6.5.6 Example Testcase

You can download a testcase from `$DENALI/example/vera/denali_vera.tar.gz`.

The example testcase utilizes DRAM to exhibit many of the common capabilities of the Denali-Vera interface.

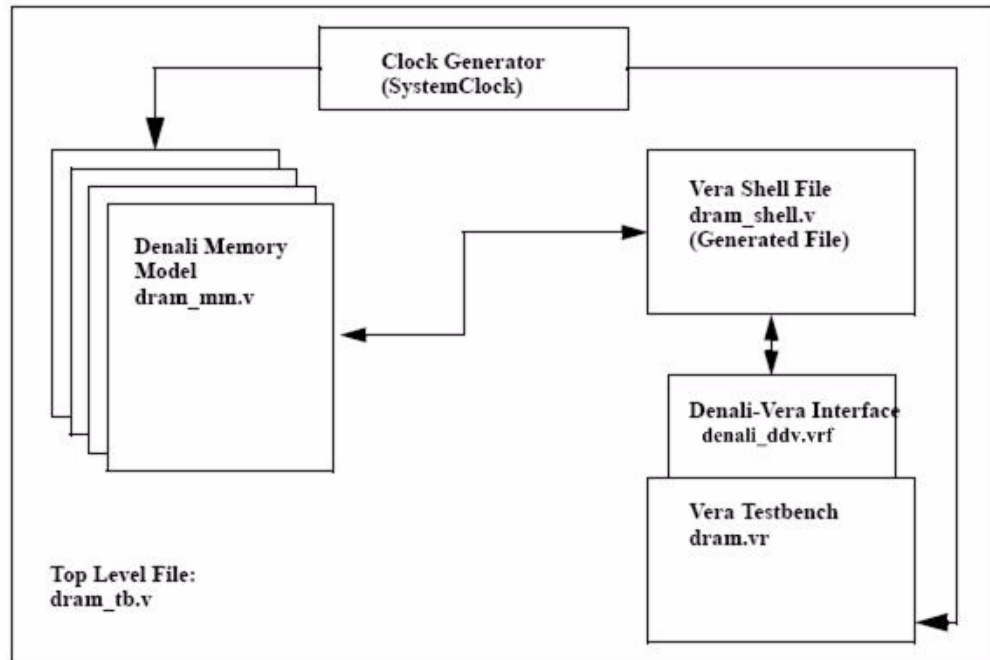


FIGURE 6-4: Diagram of Denali-Vera Example

6.6 NTB Interface

6.6.1 MMAV and NTB Overview

MMAV provides a native class-based object-oriented interface to support NTB testcases. The interface consists of NTB classes and methods that let you track memory accesses as well as perform backdoor reads and writes. You can also perform other operations that are available in the (PLI-based) Verilog procedural interface using these NTB classes.

6.6.2 Simulating with MMAV and NTB

This section provides details on simulating MMAV and NTB using Synopsys VCS simulator.

Synopsys VCS

To run simulation with MMAV and NTB using Synopsys VCS, do the following:

1. Compile the *denaliMem.vr* and *denaliMemVrIf.c* files along with the Verilog and Vera source/testbench files
2. Link in the Denali Verilog library (*denverlib.o*)

```
vcs \  
-ntb +vc+allhdrs \  
-M -P $DENALI/verilog/pli.tab \  
-CFLAGS "-DDENALI_USE_NTB=1 -I/usr/local/include -I${DENALI} -  
I${DENALI}/ddvapi -c " \  
-LDFLAGS "-rdynamic $DENALI/verilog/denverlib.o" \  
$DENALI/ddvapi/vera/denaliMemVrIf.c \  
-ntb_define DENALI_USE_NTB=1 $DENALI/ddvapi/vera/denaliMem.vr \  
+incdir+$DENALI/ddvapi/verilog \  
*.vr *.v;
```

NOTE: The *'-rdynamic'* flag used above is required only for Linux.

6.6.3 Instance and Transaction Classes

NTB memory access functions lets you track the memory references and perform read/write operations.

The main classes are:

- [“Class denaliMemInstance” on page 142](#)

- [“Class denaliMemInstanceList” on page 154](#)
- [“Class denaliMemTransaction” on page 155](#)

Each field in these classes has `get<field_name>` access methods associated with it for getting the field values. In addition to that all writable fields have `set<field_name>` for setting the field values. Since these fields are *public*, you can also access these directly.

All methods that return a status code always return a “-1” on error and a “0” on success. Unless specified otherwise, only the default constructor is available.

The `set<field_name>` and `get<field_name>` methods have the first letter of the given field name is capitalized. So if you would like to get the value of `Address`, the method to do so is `getAddress`.

Class denaliMemInstance

In order to use MMAV in the NTB environment to access memory, instantiate a `denaliMemInstance`. The `denaliMemInstance` corresponds to the memory instance either instantiated in the testbench or created by a Denali model, such as a configuration space.

The following sections describe the `denaliMemInstance` class.

Constructor

```
task new(string instName, string cbTaskName = "")
```

Arguments

Name	Type	Description
<code>instName</code>	string	The instance name.
<code>cbTaskName</code>	string	The callback task name.

Description

Creates a new instance object.

The `instName` must be a full path name and the `cbTaskName` can be null for the constructor. However, if you wish to define an explicit DPI callback task, it must be set before any callback point is added for monitoring.

NOTE: *The `cbTaskName` field is deprecated.*

Returned Value

This task returns a newly created object.

Example

```
denaliMemInstance inst;  
  
inst = new("ddr0");
```

Methods

Name	Description
getId()	Gets the Id.
getInstName()	Gets the instance name.
setCbTaskName()	Sets the DPI callback task name.
getCbTaskName()	Gets the DPI callback task name.
setCallback()	Sets callbacks on memory accesses.
write()	Writes the memory contents.
read()	Returns the memory contents.
tclEval()	Executes a Tcl command using the embedded the Tcl interpreter.
ReadCbT()	This task is called whenever a read callback occurs.
WriteCbT()	This task is called whenever a write callback occurs.
DefaultCbT()	This task is called when any enabled callback occurs.
LoadCbT()	This task is called when a load callback occurs.
LoadDoneCbT()	This task is called when a load done callback occurs.
ResetCbT()	This task is called when a reset callback occurs.
CompCbT()	This task is called when a compare callback occurs.
CompDoneCbT()	This task is called when a compare done callback occurs.
ReadEiCbT()	This task is called when an error is being injected in the current memory read operation.

getId()

Gets the id that can be used to access Denali memory.

Syntax

```
virtual function integer getId()
```

Arguments

None

Returned Value

This function returns the id.

Example

```
denaliMemInstance inst;

inst = new("ddr0");

$display("MemId = %d\n ", inst.getId());
```

getInstName()

Gets the instance name.

Syntax

```
virtual function string getInstName()
```

Arguments

None

Returned Value

This function returns the instance name.

Example

```
denaliMemInstance inst;

inst = new("ddr0");

$display("InstName = %s", inst.getInstName());
```

setCbTaskName()

Sets the DPI callback task name.

NOTE: *Denali does not recommend a user-defined DPI task.*

Syntax

```
virtual task setCbTaskName(string cbTaskName)
```

Arguments

Name	Type	Description
cbTaskName	string	The callback task name.

Returned Value

None

Example

```
denaliMemInstance inst;  
  
inst = new("ddr0");  
  
inst.setCbTaskName("myCbFunc");
```

getCbTaskName()

Gets the DPI callback task name.

NOTE: *Denali does not recommend a user-defined DPI task.*

Syntax

```
virtual function string getCbTaskName()
```

Arguments

None

Returned Value

This function returns the callback task name.

Example

```
denaliMemInstance inst;  
  
inst = new("ddr0");  
inst.setCbTaskName("myCbFunc");  
  
$display("TaskName = %s", inst.getCbTaskName());
```

setCallback()

Sets a callback on memory access.

Syntax

virtual function integer setCallback(DENALIDDVCBpointT cbRsn)

Arguments

Name	Type	Description
cbRsn	DENALIDDVCB-pointT	Callback reason.

Returned Value

This function returns 0 if successful, non-zero if failed.

Example

```
integer status;  
denaliMemInstance inst;  
  
inst = new("ddr0");  
  
status = inst.setCallback(DENALI_CB_Write);
```

write()

Writes the memory contents.

Syntax

```
virtual function integer write(var denaliMemTransaction tr)
```

Arguments

Name	Type	Description
tr	denaliMemTransaction	Contains data, address, and other relevant fields for the write operation.

Returned Value

This function returns 0 if successful, non-zero if failed.

Example

```
task writeMem(reg [63:0] addr)
{
    integer i;
    integer status;
    reg [7:0] data [*];
    denaliMemTransaction tr = new;
    tr.setAddress(addr);

    data = new[8]; // memory width is 64 bits
    for (i = 0; i < 8; i++) {
        data[i] = 'h10 + i;
    }
    tr.setData(data);
    status = mem.write(tr);
    printf("## MEM WRITE : %x -> ", tr.getAddress());
    for (i = 0; i < data.size(); i++) {
        printf("%x ", data[i]);
    }
    printf("\n");
}

program main
{
    denaliMemInstance mem;
    mem = new("ddr0");
    writeMem('h569);
}
```

read()

Returns the memory contents.

Syntax

```
virtual function integer read(var denaliMemTransaction tr)
```

Arguments

Name	Type	Description
tr	denaliMemTransaction	Contains address and other relevant fields for the read operation.

Returned Value

This function returns 0 if successful, non-zero if failed.

Example

```
task readMem(reg [63:0] addr)
{
    integer i;
    integer status;
    reg [7:0] data [*];
    denaliMemTransaction tr = new;
    tr.setAddress(addr);
    status = mem.read(tr);
    tr.getData(data);
    printf("## MEM READ : %x -> ", tr.getAddress());
    for (i = 0; i < data.size(); i++) {
        printf("%x ", data[i]);
    }
    printf("\n");
}

program main
{
    denaliMemInstance mem;
    mem = new("ddr0");
    readMem('h68);
}
```

tclEval()

Executes a Tcl command using the embedded the Tcl interpreter.

Syntax

```
virtual function integer tclEval(string cmd)
```

Arguments

Name	Type	Description
cmd	string	Tcl command.

Returned Value

This function returns 0 if successful, non-zero if failed.

Example

```
integer status;  
denaliMemInstance inst;  
  
inst = new("ddr0");  
  
status = inst.tclEval("mmsetvar tracefile -gzip denali.trc.gz");
```

ReadCbT()

This task is called whenever a read callback occurs. This happens only if the callback DENALI_CB_Read is enabled and no user-defined callback task is set.

Syntax

```
virtual task ReadCbT(var denaliMemTransaction tr)
```

Arguments

Name	Type	Description
tr	denal- iMemTransac- tion	Contains address and other relevant fields for the read operation.

Description

You can extend `denaliMemInstance` and can provide your own implementation of this task.

Refer to example shown in [“Example Testcase” on page 163](#).

Returned Value

None

WriteCbT()

This task is called whenever a write callback occurs. This happens only if the callback `DENALI_CB_Write` is enabled and no user-defined callback task is set.

Syntax

```
virtual task WriteCbT(var denaliMemTransaction tr)
```

Arguments

Name	Type	Description
tr	denaliMemTransaction	Contains data, address, and other relevant fields for the write operation.

Description

You can extend `denaliMemInstance` and can provide your own implementation of this task.

Refer to example shown in [“Example Testcase” on page 163](#).

Returned Value

None

DefaultCbT()

This task is called when any enabled callback occurs and no user-defined callback task is set.

Syntax

```
virtual task DefaultCbT(var denaliMemTransaction tr)
```

Arguments

Name	Type	Description
tr	denaliMemTransaction	Contains data, address, and other relevant fields for the operation.

Description

You can extend `denaliMemInstance` and can provide your own implementation of this task.

Refer to example shown in “Example Testcase” on page 163.

Returned Value

None

LoadCbT()

This task is called when a load callback occurs.

Syntax

```
virtual task LoadCbT(var denaliMemTransaction tr)
```

Arguments

Name	Type	Description
tr	denaliMemTransaction	Contains data, address, and other relevant fields for the operation.

Description

You can extend `denaliMemInstance` and can provide your own implementation of this task.

Returned Value

None

LoadDoneCbT()

This task is called when a load done callback occurs.

Syntax

```
virtual task LoadDoneCbT(var denaliMemTransaction tr)
```

Arguments

Name	Type	Description
tr	denaliMemTransaction	Contains data, address, and other relevant fields for the operation.

Description

You can extend `denaliMemInstance` and can provide your own implementation of this task.

Returned Value

None

ResetCbT()

This task is called when a reset callback occurs.

Syntax

```
virtual task ResetCbT(var denaliMemTransaction tr)
```

Arguments

Name	Type	Description
tr	denaliMemTransaction	Contains data, address, and other relevant fields for the operation.

Description

You can extend `denaliMemInstance` and can provide your own implementation of this task.

Returned Value

None

CompCbT()

This task is called when a compare callback occurs.

Syntax

```
virtual task CompCbT(var denaliMemTransaction tr)
```

Arguments

Name	Type	Description
tr	denaliMemTransaction	Contains data, address, and other relevant fields for the operation.

Description

You can extend `denaliMemInstance` and can provide your own implementation of this task.

Returned Value

None

CompDoneCbT()

This task is called when a compare done callback occurs.

Syntax

```
virtual task CompDoneCbT(var denaliMemTransaction tr)
```

Arguments

Name	Type	Description
tr	denaliMemTransaction	Contains data, address, and other relevant fields for the operation.

Description

You can extend `denaliMemInstance` and can provide your own implementation of this task.

Returned Value

None

ReadEiCbT()

This task is called when an error is being injected in the current memory read operation.

Syntax

```
virtual task ReadEiCbT(var denaliMemTransaction tr)
```

Arguments

Name	Type	Description
tr	denaliMemTransaction	Contains data, address, and other relevant fields for the operation.

Description

You can extend `denaliMemInstance` and can provide your own implementation of this task.

Returned Value

None

Class denaliMemInstanceList

This class is a container for all Denali memory instances instantiated by you.

NOTE: *Denali does not recommend using this class. This was used in the past to retrieve instance name at the callback points, but MMAV NTB new callback methodology has made this class obsolete.*

getInstanceFromId()

Retrieves the instance name for the specified instance id.

Syntax

```
function denaliMemInstance getInstanceFromId(integer id)
```

Arguments

Name	Type	Description
id	integer	The instance Id.

Returned Value

This function returns the memory instance object name.

Class denaliMemTransaction

This is a data structure that contains fields that are relevant for the memory access operations.

Constructor

```
function new()
```

Description

Creates a new memory transaction object of this class, which can then be used for reading/writing.

Returned Value

This function returns a memory transaction object.

Example

```
denaliMemTransaction tr;  
tr = new;
```

Fields

Name	rand (Y/N)	Type	Description
Callback	N	DENALIDVCBpointT	The callback reason.
Width	Y	integer	The width of the data in bits.
Address	Y	reg [63:0]	The address location to be read from or written to.
Data []	Y	reg [7:0]	The data to write or the data just read.
Mask []	Y	reg [7:0]	The mask to use (1 = Write).

Methods

Name	Description
getCallback()	Returns the callback reason.
getWidth()	Gets the memory width in bits.
setAddress()	Sets the address location to be read from or written to.
getAddress()	Returns the address location.
setData()	Sets the data that needs to be written.
getData()	Returns the data array being read or written.
getDataSize()	Returns the data size.
setMask()	Sets the mask to use (1 = Write) for the data being read or written
getMask()	Returns the mask used for the data.
getMaskSize()	Returns the mask size.
printInfo()	Prints the contents of the transaction object.

getCallback()

Returns the callback reason.

Syntax

```
virtual function DENALIDVCBpointT getCallback()
```

Arguments

None

Description

When the transaction object is retrieved at a callback point, this function returns the callback reason.

Returned Value

This function returns the callback reason.

Example

```
virtual task WriteCbT(var denaliMemTransaction tr)
{
    printf("Callback : %s\n", tr.getCallback());
    super.WriteCbT(tr);
}
```

getWidth()

Gets the memory width in bits.

Syntax

virtual function integer getWidth()

Arguments

None

Returned Value

This function returns the memory width.

Example

```
virtual task WriteCbT(var denaliMemTransaction tr)
{
    printf("Memory Width : %d\n", tr.getWidth());
    super.WriteCbT(tr);
}
```

setAddress()

Sets the address location to be read from or written to.

Syntax

```
virtual task setAddress(reg [63:0] Address)
```

Arguments

Name	Type	Description
Address	reg [63:0]	Specifies the address.

Returned Value

None

Example

```
denaliMemTransaction tr = new;  
tr.setAddress('h100);
```

getAddress()

Returns the address location.

Syntax

```
virtual function reg [63:0] getAddress()
```

Arguments

None

Returned Value

This function returns the address being accessed.

Example

```
printf("## MEM WRITE : %x -> ", tr.getAddress());
```

setData()

Sets the data that needs to be written.

Syntax

```
virtual task setData(reg [7:0] Data [*])
```

Arguments

Name	Type	Description
Data [*]	reg [7:0]	Specifies the data.

Returned Value

None

Example

```
tr.setData(data);  
status = mem.write(tr);
```

getData()

Returns the data array being read or written.

Syntax

```
virtual task getData(var reg [7:0] Data [*])
```

Arguments

Name	Type	Description
Data [*]	reg [7:0]	The data array.

Returned Value

This function returns the data array being read or written. The accessed data is returned in the Data argument.

Example

```
integer status;  
reg [7:0] data *;denaliMemTransaction tr = new;  
tr.setAddress('h100);  
status = mem.read(tr);  
tr.getData(data);
```

getDataSize()

Returns the data size.

Syntax

```
virtual function integer getDataSize()
```

Arguments

None

Returned Value

This function returns the size of the data array.

Example

```
virtual task WriteCbT(var denaliMemTransaction tr)  
{  
    printf("Data Size : %d\n", tr.getDataSize());  
    super.WriteCbT(tr);  
}
```

setMask()

Sets the mask to use (1 = Write) for the data being read or written.

Syntax

```
virtual task setMask(reg [7:0] Mask [*])
```

Arguments

Name	Type	Description
Mask [*]	reg [7:0]	Specifies the mask.

Returned Value

None

Example

```
integer i;
reg [7:0] mask [*];

denaliMemTransaction tr = new;
tr.setAddress(`h40);
mask = new[8]; // our memory width is 64 bits
for (i = 0; i < 8; i++) {
    mask[i] = `b01010101;
}
tr.setMask(mask);
```

getMask()

Returns the mask used for the data in the `Mask` argument.

Syntax

```
virtual task getMask(var reg [7:0] Mask [*])
```

Arguments

Name	Type	Description
Mask [*]	integer	Specifies the data mask.

Returned Value

This function returns the mask array being read or written. The accessed mask is returned in the `Mask` argument.

Example

```
virtual task WriteCbT(var denaliMemTransaction tr)
{
    integer i;
    reg [7:0] mask [*];

    tr.getMask(mask);
    printf("### MASK : ");
    for (i = 0; i < mask.size(); i++) {
        printf("%x ", mask[i]);
    }
    super.WriteCbT(tr);
}
```

getMaskSize()

Returns the mask size.

Syntax

```
virtual function integer getMaskSize()
```

Arguments

None

Returned Value

This function returns the size of the mask array.

Example

```
virtual task WriteCbT(var denaliMemTransaction tr)
{
    printf("Mask Size : %d\n", tr.getMaskSize());
    super.WriteCbT(tr);
}
```

printInfo()

Prints the contents of the transaction object.

Syntax

```
virtual task printInfo(integer arrayDepth = 32)
```

Arguments

Name	Type	Description
arrayDepth	integer	Specifies the maximum number of array elements that need to be printed.

Returned Value

This function returns 0 if successful, non-zero if failed.

Example

```
virtual task DefaultCbT(var denaliMemTransaction tr)
{
    tr.printInfo();
    super.DefaultCbT(tr);
}
```

6.6.4 Processing Callbacks

MMAV provides a way to generate callbacks to your NTB testbench on memory accesses.

MMAV NTB callback interface provides the facilities for model callback initialization and handling.

For this, you should add the relevant callbacks to the device. You can setup callback functions by extending `denaliMemInstance` (the instance class) and overloading the built-in callback functions (specified by not setting the `cbTaskName` parameter). You can overload a virtual task per callback reason in the class.

Refer to [“Example Testcase” on page 163](#) for a detailed example on callback processing.

6.6.5 Example Testcase

The following example shows the instantiation of a DDR memory and backdoor reads and writes. It also shows how the `denaliMemInstance` class is extended to handle callbacks.

```

#include "denaliMemTypes.vrh"

task writeMem(reg [63:0] addr)
{
    integer i;
    integer status;
    reg [7:0] data [*];

    denaliMemTransaction tr = new;
    tr.setAddress(addr);

    data = new[8]; // memory width is 64 bits
    for (i = 0; i < 8; i++) {
        data[i] = 'h10 + i;
    }

    tr.setData(data);
    status = mem.write(tr);

    printf("## MEM WRITE : %x -> ", tr.getAddress());
    for (i = 0; i < data.size(); i++) {
        printf("%x ", data[i]);
    }
    printf("\n");
}

task readMem(reg [63:0] addr)
{
    integer i;
    integer status;
    reg [7:0] data [*];

    denaliMemTransaction tr = new;
    tr.setAddress(addr);
    status = mem.read(tr);
    tr.getData(data);

    printf("## MEM READ : %x -> ", tr.getAddress());
    for (i = 0; i < data.size(); i++) {
        printf("%x ", data[i]);
    }
    printf("\n");
}

```

continued...

```

...continued

class MyDenaliMemInstance extends denaliMemInstance
{
    task new(string instName)
    {
        super.new(instName);
    }

    virtual task WriteCbT(var denaliMemTransaction tr)
    {
        printf("*****\n");
        tr.printInfo();
        printf("*****\n");
        WriteCbT = super.WriteCbT(tr);
    }

    virtual task ReadCbT(var denaliMemTransaction tr)
    {
        printf("*****\n");
        tr.printInfo();
        printf("*****\n");
        ReadCbT = super.ReadCbT(tr);
    }
}

program main
{
    integer status;
    MyDenaliMemInstance mem;
    denaliMemTransaction tr;
    status = denaliMemInit();
    if (status == -1) {
        error("Denali DDV-MMAV initialization failed. Cannot continue
        ...\n");
        exit(1);
    }

    mem = new("simple_tb.simple_mem.storage");
    status = mem.setCallback(DENALI_CB_Read);
    status = mem.setCallback(DENALI_CB_Write);

    writeMem('h68);
    readMem('h68);

    while (1) {
        @(posedge CLOCK);
    }

    @(posedge CLOCK);
}

```

6.7 SystemVerilog Interface

This section provides details on how to use MMAV in a SystemVerilog testbench.

6.7.1 MMAV and SystemVerilog Overview

MMAV provides a native class-based object-oriented interface to support SystemVerilog testbenches. The interface consists of SV classes and methods that let you track memory accesses as well as perform backdoor reads and writes. Other operations that are found in the (PLI-based) verilog procedural interface can also be performed with these SV classes.

6.7.2 Simulating with MMAV and SystemVerilog

This section describes the simulation steps for some of the commonly used Verilog simulators.

Mentor Graphics Questa

To run simulation with MMAV and SV using Mentor Graphics Questa, do the following:

1. Set an environment variable `$DENALI` to the root directory of your Denali installation.
2. Setup path for your SystemVerilog simulator.
3. Generate your MMAV model shell file using PureView. For details, refer to [“Using the PureView Graphical Tool” on page 11](#).
4. Create your *user.sv* testbench.
5. Compile the SV files. The `-dpiheader` option creates the header file that is used in the C compilation below.

```
vlog \  
+incdir+$DENALI/ddvapi/sv -dpiheader denaliMemSvIf.h \  
$DENALI/ddvapi/sv/denaliMem.sv *.sv *.v
```

6. Compile/Link the C files. These files contain the code that translates C data structures to SV and the vice-versa.

```
gcc -c -g -fPIC -I$MTI_HOME/include -I. -I$DENALI \  
-I$DENALI/ddvapi $DENALI/ddvapi/sv/denaliMemSvIf.c
```

7. Create the shared object.

```
gcc -shared -o denaliMemSvIf.so \  
$DENALI/mtipli.so
```

8. Delete all the unnecessary files.

```
rm -f denaliMemSvIf.h denaliMemSvIf.o
```

9. Run the simulation. The `'-svlib <lib>'` option loads the interface shared object that you just created. The `-dpioutoftheblue 1` option lets you call SV exported tasks from

the C code that was invoked via PLI 1.0. The `-pli <lib>` options loads the Denali library containing the C model.

```
vsim -c top -sv_lib denaliMemSvIf -do "run -all; quit" -pli \  
$DENALI/mtipli.so -dpioutoftheblue 1
```

Cadence NC-Verilog

To run simulation with MMAV and SV using Cadence NC-Verilog, do the following:

1. Set an environment variable `$DENALI` to the root directory of your Denali installation.
2. Setup path for your SystemVerilog simulator.
3. Generate your MMAV model shell file using PureView. For details, refer to [“Using the PureView Graphical Tool” on page 11](#).
4. Create your `user.sv` testbench.

Compile the SV files. The `-dpiheader` option creates the header file that is used in the

```
C compilation below.ncverilog +sv \  
+elaborate \  
+ncdpiheader+denaliMemSvIf.h \  
+licq \  
+define+DENALI_SV_NC \  
+incdir+$DENALI/ddvapi/sv \  
$DENALI/ddvapi/sv/denaliMem.sv
```

5. Compile the C files. These files contain the code that translates C data structures to SV and the vice-versa.

```
gcc -c -g -fPIC -DDENALI_SV_NC=1 \  
-I$CDS_TOOLS/include -I. -I$DENALI \  
-I$DENALI/ddvapi \  
$DENALI/ddvapi/sv/denaliMemSvIf.c -m32
```

6. Link the C files.

```
ld -G -o denaliMemSvIf.so denaliMemSvIf.o $DENALI/verilog/libden-  
pli.so -melf_i386
```

7. Create the shared object.

```
gcc -shared -o denaliMemSvIf.so \  
$DENALI/mtipli.so
```

8. Run the simulation.

```
ncverilog +loadpli1=$DENALI/verilog/libdenpli.so:den_PLIPtr \  
+access+rw \  
+nbasync \  
+sv \  
+sv_lib=$PROJ_HOME/run/denaliMemSvIf.so \  
+incdir+$DENALI/ddvapi/sv \  
+licq \  
+define+DENALI_SV_NC \  
*.sv *.v
```

Synopsys VCS

To run simulation with MMAV and SV using Synopsys VCS, do the following:

1. Set an environment variable `$DENALI` to the root directory of your Denali installation.
2. Setup path for your SystemVerilog simulator.
3. Generate your MMAV model shell file using PureView. For details, refer to [“Using the PureView Graphical Tool” on page 11](#).
4. Create your `user.sv` testbench.

```
Compile the SV files. vcs -CFLAGS "-DDENALI_SV_VCS=1 -I${DENALI} \
-I${DENALI}/ddvapi -g -c" \
-sverilog +vcs+lic+wait \
-ntb_opts svp -ntb_opts rvm -ntb_opts dtm \
-Mupdate -P ${DENALI}/verilog/pli.tab \
-LDFLAGS "-rdynamic ${DENALI}/verilog/denverlib.o" \
-debug_pp
+ntb_enable_solver_trace=0 \
+dmpprof \
+define+DENALI_SV_VCS \
+incdir+${DENALI}/ddvapi/sv \
${DENALI}/ddvapi/sv/denaliMemSvIf.c \
${DENALI}/ddvapi/sv/denaliMem.sv \
*.sv *.v
```

5. Run the simulation.

```
./simv -l vcs.log
```

6.7.3 Configuration Register and Memory Access

SystemVerilog memory access functions are applicable to all Denali Verification IP products. These functions let you track the memory references and perform read/write operations.

The main classes are:

- Class `denaliMemInstance`
- Class `denaliMemInstanceList`
- Class `denaliMemTransaction`

Each field in these classes has `get<field_name>` access methods associated with it for getting the field values. In addition to that all writable fields have `set<field_name>` for setting the field values. Since these fields are *public*, you can also access these directly.

All methods that return a status code always return a “-1” on error and a “0” on success.

The `set<field_name>` and `get<field_name>` methods have the first letter of the given field name is capitalized. So if you would like to get the value of `Address`, the method to do so is `getAddress`.

Some fields are marked `rand` so that the SV `randomize` function can generate values for them.

Class `denaliMemInstance`

In SystemVerilog environment to access memory, instantiate a `denaliMemInstance`. The `denaliMemInstance` corresponds to the memory instance either instantiated in the test-bench or created by the model, such as a configuration space.

The following sections describe the `denaliMemInstance` class.

Constructor

```
function new(string instName, string cbFuncName = "") ;
```

Fields

Name	rand (Y/N)	Type	Description
<code>instName</code>	N	string	The instance name.
<code>cbFuncName</code>	N	string	The callback function name.

The `instName` must be a full path name and the `cbFuncName` can be null for the constructor. However, if you wish to define an explicit DPI callback function, it must be set before any callback point is added for monitoring.

NOTE: *The `cbFuncName` field is deprecated.*

Methods

Name	Description
<code>new()</code>	Creates a new instance object.
<code>getInstName()</code>	Gets the instance name.
<code>getId()</code>	Gets the id.
<code>getSize()</code>	Retrieves the size of memories.
<code>getWidth ()</code>	Retrieves the width of memories.
<code>setCbFuncName()</code>	Sets the DPI callback function name.
<code>getCbFuncName()</code>	Gets the DPI callback function name.
<code>setCallback()</code>	Sets a callback on memory access.
<code>write()</code>	Writes the memory contents.
<code>read()</code>	Returns the memory contents.

Name	Description
tclEval()	Executes a Tcl command using the embedded the Tcl interpreter.
tclEvalGetResult()	Executes a Tcl command using the embedded Tcl interpreter and returns the result in a string parameter.
setBackdoorCb-Mode()	This function is called with a parameter value of 0 to turn off backdoor access callbacks. Setting it to 1 enables them.
ReadCbF()	This function is called whenever a read callback occurs.
WriteCbF()	This function is called whenever a write callback occurs.
DefaultCbF()	This function is called when any enabled callback occurs and the callback function is not set.
LoadCbF()	This function is called when a load callback occurs.
LoadDoneCbF()	This function is called when a load done callback occurs.
ResetCbF()	This function is called when a reset callback occurs.
CompCbF()	This function is called when a compare callback occurs.
CompDoneCbF()	This function is called when a compare done callback occurs.
ReadEiCbF()	This function is called when an error is being injected in the current memory read operation.

new()

Creates a new instance object.

Syntax

```
function new(string instName, string cbFuncName = "") ;
```

Arguments

Name	Type	Description
instName	string	The instance name.
cbFuncName	string	The callback name.

Description

Creates a new packet to be initiated from the specified instance and returns a packet handle pointing to the newly created empty packet.

Returned Value

This function returns 0 if successful, non-zero if failed.

Example

```
denaliMemInstance inst;
inst = new("i0");
```

getId()

Gets the id.

Syntax

```
virtual function int getId() ;
```

Returned Value

This function returns 0 if successful, non-zero if failed.

Example

```
denaliMemInstance inst;  
inst = new("i0");  
$display("MemId = %d\n ", inst.getId());
```

getInstName()

Gets the instance name.

Syntax

```
function string getInstName() ;
```

Returned Value

This function returns 0 if successful, non-zero if failed.

Example

```
denaliMemInstance inst;  
inst = new("i0");  
$display("InstName = %s", inst.getInstName());
```

getSize()

Retrieves the size of memories.

Syntax

```
virtual function longint unsigned getSize() ;
```

Returned Value

This function returns 0 if successful, non-zero if failed.

Example

```
denaliMemInstance inst;  
inst = new("i0");  
$display("MemSize = %d\n ", inst.getSize());
```

getWidth()

Retrieves the width of memories.

Syntax

```
virtual function int unsigned getWidth() ;
```

Returned Value

This function returns 0 if successful, non-zero if failed.

Example

```
denaliMemInstance inst;  
inst = new("i0");  
$display("MemWidth = %d\n ", inst.getWidth());
```

getCbFuncName()

Gets the DPI callback function name.

NOTE: *Denali does not recommend the use of user-defined DPI function.*

Syntax

```
function string getCbFuncName() ;
```

Returned Value

This function returns 0 if successful, non-zero if failed.

Example

```
denaliMemInstance inst;  
inst = new("i0");  
inst.setCbFuncName("myCbFunc");  
$display("FuncName = %s", inst.getCbFuncName());
```

setCallback()

This function is used to set a callback on memory access.

Syntax

```
function integer setCallback(DENALIDDVCBpointT cbRsn) ;
```

Arguments

Name	Type	Description
cbRsn	DENALIDDVCBpointT	The callback reason.

Returned Value

This function returns 0 if successful, non-zero if failed.

Example

```
int status;  
denaliMemInstance inst;  
  
inst = new("i0");  
  
status = inst.setCallback(DENALI_CB_Write);
```

write()

This function writes the memory contents.

Syntax

```
function int write(ref denaliMemTransaction trans);
```

Arguments

Name	Type	Description
trans	denaliMemTransaction	Contains data, address, and other relevant fields for the write operation.

Returned Value

This function returns 0 if successful, non-zero if failed.

Example

```
function void writeData(reg [63:0] addr);
    reg [7:0] data [];
    denaliMemTransaction trans;

    trans = new;
    trans.setAddress(addr);
    data = new[4];
    data[0] = 'h01;
    data[1] = 'h34;
    data[2] = 'h78;
    data[3] = 'h90;
    trans.setData(data);
    assert(inst.write(trans) == 0);
endfunction

denaliMemInstance inst;
inst = new("i0");
inst.writeData('h569);
```

read()

This function returns the memory contents.

Syntax

```
function int read(ref denaliMemTransaction trans);
```

Arguments

Name	Type	Description
trans	denaliMemTransaction	Contains address and other relevant fields for the read operation.

Returned Value

This function returns 0 if successful, non-zero if failed.

Example

```
function void readData(reg [63:0] addr);
    reg [7:0] data [];
    int status;
    denaliMemTransaction trans;

    trans = new;
    trans.setAddress(addr);
    assert(inst.read(trans) == 0)
    trans.getData(data);
    $display("Data = %x", data);
endfunction

denaliMemInstance inst;
inst = new("i0");
inst.readData('h569);
```

tclEval()

Executes a Tcl command using the embedded the Tcl interpreter.

Syntax

```
function int tclEval(string cmd);
```

Arguments

Name	Type	Description
cmd	string	Tcl command.

Returned Value

This function returns 0 if successful, non-zero if failed.

Example

```
denaliMemInstance inst;
inst = new("i0");
assert(inst.tclEval("mmsetvar tracefile -gzip denali.trc.gz") == 0);
```

tclEvalGetResult()

Executes a Tcl command using the embedded Tcl interpreter and returns the result in a string parameter.

Syntax

```
function int tclEvalGetResult(string cmd, output string result, input
    int resultSize = 1024);
```

Arguments

Name	Type	Description
cmd	string	Tcl command.
result	string	Tcl evaluation result.
resultsize	int	The maximum result size.

Returned Value

This function returns 0 if successful, non-zero if failed.

Example

```
denaliMemInstance inst;
string res;
int status;
inst = new("i0");
status = inst.tclEvalGetResult("mmsetvar tracefile -gzip
denali.trc.gz", res);
$display("status = %d / res = %s", status, res);
```

setBackdoorCbMode()

This function is called with a parameter value of 0 to turn off backdoor access callbacks. Setting it to 1 enables them.

NOTE: *If the callback processing testbench code calls any Denali MMAV backdoor method that causes another access callback to occur at the same time and in the same delta cycle, some simulators may have trouble handling this. To ensure that the backdoor method can be called from within another callback, Denali recommends turning off backdoor access callbacks for that memory instance.*

Syntax

```
virtual function int setBackdoorCbMode(bit onOrOff);
```

Returned Value

This function returns 0 if successful, non-zero if failed.

Example


```
denaliMemInstance inst;
inst = new("testbench.i1");
// this will turn off backdoor access cb's for inst
assert(inst.setBackdoorCbMode(0));
```

ReadCbF()

This function is called whenever a read callback occurs. This happens only if the callback `DENALI_CB_Read` is enabled and the callback function is not set.

Syntax

```
virtual function int ReadCbF(ref denaliMemTransaction trans) ;
```

Arguments

Name	Type	Description
trans	denaliMemTransaction	Contains address and other relevant fields for the read operation.

Description

You can extend `denaliMemInstance` and can provide your own implementation of this method.

Returned Value

This function returns 0 if successful, non-zero if failed.

Example

```
virtual function int ReadCbF(ref denaliMemTransaction trans);
    void'(trans.printInfo());
    return super.ReadCbF(trans);
endfunction
```

WriteCbF()

This function is called whenever a write callback occurs. This happens only if the callback `DENALI_CB_Write` is enabled and the callback function is not set.

Syntax

```
virtual function int WriteCbF(ref denaliMemTransaction trans) ;
```

Arguments

Name	Type	Description
trans	denaliMemTransaction	Contains data, address, and other relevant fields for the write operation.

Description

You can extend `denaliMemInstance` and can provide your own implementation of this method.

Returned Value

This function returns 0 if successful, non-zero if failed.

Example

```
virtual function int WriteCbF(ref denaliMemTransaction trans);
    void'(trans.printInfo());
    return super.WriteCbF(trans);
endfunction
```

DefaultCbF()

This function is called when any enabled callback occurs and the callback function is not set.

Syntax

```
virtual function int DefaultCbF(ref denaliMemTransaction trans) ;
```

Arguments

Name	Type	Description
trans	denaliMemTransaction	Contains data, address, and other relevant fields for the operation.

Description

You can extend `denaliMemInstance` and can provide your own implementation of this method.

Returned Value

This function returns 0 if successful, non-zero if failed.

Example

```
virtual function int DefaultCbF(ref denaliMemTransaction trans);
    void'(trans.printInfo());
    return super.DefaultCbF(trans);
endfunction
```

LoadCbF()

This function is called when a load callback occurs.

Syntax

```
virtual function int LoadCbF(ref denaliMemTransaction trans) ;
```

Arguments

Name	Type	Description
trans	denal-iMemTrans-action	Contains data, address, and other relevant fields for the operation.

Description

You can extend `denaliMemInstance` and can provide your own implementation of this method.

Returned Value

This function returns 0 if successful, non-zero if failed.

Example

```
virtual function int LoadCbF(ref denaliMemTransaction trans);
    void'(trans.printInfo());
    return super.LoadCbF(trans);
endfunction
```

LoadDoneCbF()

This function is called when a load done callback occurs.

Syntax

```
virtual function int LoadDoneCbF(ref denaliMemTransaction trans) ;
```

Arguments

Name	Type	Description
trans	denaliMemTransaction	Contains data, address, and other relevant fields for the operation.

Description

You can extend `denaliMemInstance` and can provide your own implementation of this method.

Returned Value

This function returns 0 if successful, non-zero if failed.

Example

```
virtual function int LoadDoneCbF(ref denaliMemTransaction trans);  
    void'(trans.printInfo());  
    return super.LoadDoneCbF(trans);  
endfunction
```

ResetCbF()

This function is called when a reset callback occurs.

Syntax

```
virtual function int ResetCbF(ref denaliMemTransaction trans) ;
```

Arguments

Name	Type	Description
trans	denaliMemTransaction	Contains data, address, and other relevant fields for the operation.

Description

You can extend `denaliMemInstance` and can provide your own implementation of this method.

Returned Value

This function returns 0 if successful, non-zero if failed.

Example

```
virtual function int ResetCbF(ref denaliMemTransaction trans);
    void'(trans.printInfo());
    return super.ResetCbF(trans);
endfunction
```

CompCbF()

This function is called when a compare callback occurs.

Syntax

```
virtual function int CompCbF(ref denaliMemTransaction trans);
```

Arguments

Name	Type	Description
trans	denal- iMemTransac- tion	Contains data, address, and other relevant fields for the operation.

Description

You can extend `denaliMemInstance` and can provide your own implementation of this method.

Returned Value

This function returns 0 if successful, non-zero if failed.

Example

```
virtual function int CompCbF(ref denaliMemTransaction trans);
    void' (trans.printInfo());
    return super.CompCbF(trans);
endfunction
```

CompDoneCbF()

This function is called when a compare done callback occurs.

Syntax

```
virtual function int CompDoneCbF(ref denaliMemTransaction trans) ;
```

Arguments

Name	Type	Description
trans	denaliMemTransaction	Contains data, address, and other relevant fields for the operation.

Description

You can extend `denaliMemInstance` and can provide your own implementation of this method.

Returned Value

This function returns 0 if successful, non-zero if failed.

Example

```
virtual function int CompDoneCbF(ref denaliMemTransaction trans);
    void' (trans.printInfo());
    return super.CompDoneCbF(trans);
endfunction
```

ReadEiCbF()

This function is called when an error is being injected in the current memory read operation.

Syntax

```
virtual function int ReadEiCbF(ref denaliMemTransaction trans) ;
```

Arguments

Name	Type	Description
trans	denaliMemTransaction	Contains data, address, and other relevant fields for the operation.

Description

You can extend `denaliMemInstance` and can provide your own implementation of this method.

The `ReadEiCbF()` callback function works together with the `$mmerrinject()` function and is primarily used in the DRAM for ECC validation.

For details on the the usage of the `$mmerrinject()` function, refer to MMAV User's Guide located at *\$DENALI/doc/mmap/mmapUserGuide.pdf*.

Here is an example that shows how to use `$mmerrinject()` function:

```
denaliMemInstance inst;  
inst = new("i0");  
assert(inst.tclEval("mmerrinject id -seed 12 -reads 1000 1200 -bits 1 2  
4 -percent 80 15 5") == 0);
```

In this, the model randomly injects errors between 1000 and 1200 read operations with 1 bit error injection 80% of the time, 2 bit error injection 15% of the time, and 4 bit error injection 5% of the time. A particular Nth read operation between the range 1000-1200 is random as well as for which particular bits get flipped.

You can use a few variations of the `$mmerrinject()` function call. For example, if you do not need to specify a range, then you can only specify “-reads 1000” to inject an error every 1000 read operations for instance.

NOTE: The `ReadEiCbF()` function is **only** true and triggered for the Nth read operation for which an error injection takes place. The Data field returns the value and the Mask in this case `!=NULL`. `Mask[i] = 1` means `Data[i]` is true memory content and `Mask[i]=0` means `Data[i]` is flipped (0->1, or 1->0).

Returned Value

This function returns 0 if successful, non-zero if failed.

Example

```

virtual function int ReadEiCbF(ref denaliMemTransaction trans);
    void'(trans.printInfo());
    return super.ReadEiCbF(trans);
endfunction

```

Class denaliMemInstanceList

This class is a container for all Denali memory instances instantiated by you.

NOTE: *Denali does not recommend using this class. This was used in the past to retrieve instance name at the callback points, but SystemVerilog new callback methodology has made this class obsolete.*

getInstanceFromId()

Retrieves the instance name for the specified instance id.

Syntax

```
static function denaliMemInstance getInstanceFromId(integer id) ;
```

Arguments

Name	Type	Description
id	integer	The instance Id.

Returned Value

This function returns 0 if successful, non-zero if failed.

Class denaliMemTransaction

This is a data structure that contains fields that are relevant for the memory access operations.

Constructor

```
function new() ;
```


Fields

Name	rand (Y/N)	Type	Description
Callback	N	DENALIDDVCB-pointT	The callback reason.
Width	Y	integer	The width of the data in bits.
Address	Y	reg [63:0]	The address location to be read from or written to.
Data []	Y	reg [7:0]	The data to write or the data just read.
Mask []	Y	reg [7:0]	The mask to use (1 = Write).

Methods

Name	Description
new()	Creates a memory transaction object.
printInfo()	Prints the contents of the transaction object.
AVM Methods	
clone()	Returns a copy of the AVM object.
comp()	Compares two transaction objects.
convert2string()	Returns a string representation of the object.
VMM Methods	
psdisplay()	Returns an image of the current value of the transaction or data described by this instance in a readable format as a string.
is_valid()	Checks if the current value of the transaction or data described by this instance is valid and error-free, according to the optionally specified kind or format. Note: This is not implemented yet.
copy()	Copies the current value of the object instance to the specified object instance. If no target object instance is specified, a new instance is allocated.
allocate()	Allocates a new instance of the same type as the object instance.
compare()	Compares the current value of the object instance with the current value of the specified object instance, according to the specified kind. Returns TRUE (i.e., non-zero) if the value is identical.
byte_pack()	Packs the name of the action descriptor into the specified dynamic array of bytes, starting at the specified offset in the array and ending with a byte set to 8'h00. Note: This is not implemented yet.
byte_unpack()	Unpacks the name of the action descriptor from the specified offset in the specified dynamic array until a byte set to 8'h00, the specified number of bytes have been unpacked or the end of the array is encountered, whichever comes first. Note: This is not implemented yet.
byte_size()	Returns the number of bytes required to pack the content of the descriptor. Note: This is not implemented yet.

new()

Creates a new memory transaction object.

Syntax

```
function new() ;
```

Description

Creates a new memory transaction object of this class, which can then be used for reading/writing.

Returned Value

This function returns 0 if successful, non-zero if failed.

Example

```
denaliMemTransaction trans;  
trans = new;
```

printInfo()

Prints the contents of the transaction object.

Syntax

```
virtual function integer printInfo(integer arrayDepth = 32) ;
```

Arguments

Name	Type	Description
arrayDepth	integer	Specifies the maximum number of array elements that need to be printed.

Returned Value

This function returns 0 if successful, non-zero if failed.

Example

```
virtual function int DefaultCbF(ref denaliMemTransaction tr);  
    assert(tr.printInfo() == 0);  
    return super.DefaultCbF(tr);  
endfunction
```

clone()

Returns a copy of the AVM object.

Syntax

```
virtual function denaliMemTransaction clone() ;
```

Returned Value

This function returns 0 if successful, non-zero if failed.

Example

```
static denaliMemTransaction trQ [$];

virtual function int DefaultCbF(ref denaliMemTransaction tr);
    trQ.push_front(tr.clone());
    return super.DefaultCbF(tr);
endfunction
```

comp()

Compares two transaction objects.

Syntax

```
virtual function bit comp(input denaliMemTransaction item) ;
```

Arguments

Name	Type	Description
item	denaliMemTransaction	Specifies the object against which the current object is compared.

Returned Value

This function returns 0 if successful, non-zero if failed.

Example

```
static denaliMemTransaction trQ [$];

virtual function int DefaultCbF(ref denaliMemTransaction tr);
    if ((trQ.size == 1) && (tr.comp(trQ[0]))) begin
        $display("Match found!");
    end
    return super.DefaultCbF(tr);
endfunction
```

convert2string()

Returns a string representation of the object.

Syntax

```
virtual function string convert2string() ;
```

Returned Value

This function returns 0 if successful, non-zero if failed.

Example

```
virtual function int DefaultCbF(ref denaliMemTransaction tr);  
    tr.convert2string();  
    return super.DefaultCbF(tr);  
endfunction
```

psdisplay()

Returns an image of the current value of the transaction or data described by this instance in a readable format as a string.

Syntax

```
virtual function string psdisplay(string prefix = "");
```

Description

The string may contain newline characters to split the image across multiple lines. Each line of the output must be prefixed with the specified prefix.

Returned Value

Returns an image of the current value of the transaction or data described by this instance in a readable format as a string.

is_valid()

NOTE: *This function is not implemented yet.*

Checks if the current value of the transaction or data described by this instance is valid and error-free, according to the optionally specified kind or format.

Syntax

```
virtual function bit is_valid(bit silent = 1, int kind = -1);
```

Description

Checks if the current value of the transaction or data described by this instance is valid and error-free, according to the optionally specified kind or format. Returns TRUE (i.e., non-zero) if the content of the object is valid. Returns FALSE otherwise. The meaning (and use) of the kind argument is descriptor-specific and defined by the user-extension of this method.

If silent is TRUE (i.e., non-zero), no error or warning messages are issued if the content is invalid. If silent is FALSE, warning or error messages may be issued if the content is invalid.

Returned Value

This function returns 0 if successful, non-zero if failed.

copy()

Copies the current value of the object instance to the specified object instance. If no target object instance is specified, a new instance is allocated.

Syntax

```
virtual function vmm_data copy(vmm_data to = null);
```

Description

Copies the current value of the object instance to the specified object instance. If no target object instance is specified, a new instance is allocated. Returns a reference to the target instance. Note that the following trivial implementation will not work. Constructor copying is a shallow copy. The objects instantiated in the object (such as those referenced by the log and notify properties) are not copied and both copies will share references to the same service interfaces. Furthermore, it will not properly handle the case when the to argument is not null.

Returned Value

This function returns a reference to the target instance.

allocate()

Allocates a new instance of the same type as the object instance.

Syntax

```
virtual function vmm_data allocate();
```

Description

Allocates a new instance of the same type as the object instance. Returns a reference to the new instance. Useful to implement class factories to create instances of user-defined derived class in generic code written using the base class type.

Returned Value

This function returns 0 if successful, non-zero if failed.

compare()

Compares the current value of the object instance with the current value of the specified object instance, according to the specified kind. Returns TRUE (i.e., non-zero) if the value is identical.

Syntax

```
virtual function bit compare(input vmm_data to, output string diff,  
    input int kind = -1)'
```

Description

Compares the current value of the object instance with the current value of the specified object instance, according to the specified `kind`. Returns TRUE (i.e., non-zero) if the value is identical. If the value is different, FALSE is returned and a descriptive text of the first difference found is returned in the specified string variable. The `kind` argument may be used to implement different comparison functions (e.g., full compare, comparison of rand properties only, comparison of all properties physically implemented in a protocol and so on.).

Returned Value

This function returns 0 if the value is identical.

byte_pack()

NOTE: *This function is not implemented yet.*

Packs the name of the action descriptor into the specified dynamic array of bytes, starting at the specified offset in the array and ending with a byte set to 8'h00.

Syntax

```
virtual function int unsigned byte_pack(ref logic [7:0] bytes[],  
    input int unsigned offset = 0, input int kind = -1);
```

Description

The default implementation packs the name of the action descriptor into the specified dynamic array of bytes, starting at the specified offset in the array and ending with a byte set to 8'h00. The array is resized appropriately.

Returned Value

This function returns the number of bytes added to the array.

byte_unpack()

NOTE: *This function is not implemented yet.*

Unpacks the name of the action descriptor from the specified offset in the specified dynamic array until a byte set to 8'h00, the specified number of bytes have been unpacked or the end of the array is encountered, whichever comes first.

Syntax

```
virtual function int unsigned byte_unpack(const ref logic [7:0]
    bytes[], input int unsigned offset = 0, input int len = -1, input
    int kind = -1);
```

Description

The default implementation unpacks the name of the action descriptor from the specified offset in the specified dynamic array until a byte set to 8'h00, the specified number of bytes have been unpacked or the end of the array is encountered, which-ever comes first.

Returned Value

This function returns the number of bytes unpacked.

byte_size()

NOTE: *This function is not implemented yet.*

Returns the number of bytes required to pack the content of the descriptor.

Syntax

```
virtual function int unsigned byte_size(int kind = -1);
```

Description

This method will be more efficient than `vmm_data::byte_pack()` for simply knowing how many bytes are required by the descriptor because no packing is actually done.

If the data can be interpreted or packed in different ways, the `kind` argument can be used to specify which interpretation or packing to use.

Returned Value

This function returns the number of bytes required to pack the content of this descriptor.

Callback Processing

Denali provides a way to generate callbacks to your SystemVerilog testbench on memory accesses.

The SystemVerilog callback interface provides the facilities for model callback initialization and handling.

For this, you should add the relevant callbacks to the device. You can setup callback functions by extending `denaliMemInstance` (the instance class) and overload the built-in callback functions (specified by not setting the `cbFuncName` parameter). You can overload a virtual function per callback reason in the class. Refer the example testcase below.

Example Testcase

```
module top;
  import DenaliSvMem::*;

  /* extend the denaliMemInstance
  */
  class MyMemInstance extends denaliMemInstance;
    function new (string instName);
      super.new (instName);
    endfunction

    /* overload the write callback function
    */

    virtual function int WriteCbF (ref denaliMemTransaction trans);
      $display ("WriteCbF (");
      void'(trans.printInfo());
      return super.WriteCbF (trans);
    endfunction

    /* overload the read callback function
    */

    virtual function int ReadCbF (ref denaliMemTransaction trans);
      $display ("ReadCbF (");
      void'(trans.printInfo ());
      return super.ReadCbF (trans);
    endfunction

  endclass

                                continued...
```

```

...continued
MyMemInstance inst;

    task init_tb();
        assert(denaliMemInit() == 0);

        // instantiate MyMemInstance for the momory model/configuration
space
        inst = new("top.i0");
        // set the read and write callback points
        assert(inst.setCallback(DENALI_CB_Write) == 0);
        assert(inst.setCallback(DENALI_CB_Read) == 0);
    endtask

    initial
        begin
            init_tb();
            writeData('h569);
            readData('h569);
        end

    function void writeData(reg [63:0] addr);
        reg [7:0] data [];
        denaliMemTransaction trans;
        trans = new;
        trans.setAddress(addr);
        data = new[4];
        data[0] = 'h01;
        data[1] = 'h34;
        data[2] = 'h78;
        data[3] = 'h90;
        trans.setData(data);
        assert(inst.write(trans) == 0);
    endfunction

    function void readData(reg [63:0] addr);
        reg [7:0] data [];
        int status;
        denaliMemTransaction trans;
        trans = new;
        trans.setAddress(addr);
        assert(inst.read(trans) == 0);
        trans.getData(data);
        $display("Data = %x", data);
    endfunction

endmodule

```


A Getting Technical Support

Denali has a quick and easy procedure for resolving technical issues with its products. If you suspect a problem with Denali tools while you are simulating, follow these three steps. If the problem is not related to simulation, go to step 3.

Step 1: Check the Denali MMAV FAQ

You can find the Denali MMAV FAQ at: <http://www.denali.com/support>. Check here first as many commonly asked questions have been added to the FAQ knowledge database. If the FAQ does not help, proceed to Step 2.

Step 2: Generating Simulation Results

The History file and Trace files are essential to reproduce the behavior of your models. To generate the appropriate files, simply run the simulation after un-commenting the following lines in your local *.denalirc* file:

```
HistoryFile <historyfile.his>
HistoryDebug On
TraceFile <tracefile.trc>
```

Note, you can restrict this history and trace files to certain memory instances. To do this, there are a couple settings in the *.denalirc* file that control this:

TracePattern

TracePattern allows you to limit the size of your trace file (see above) by limiting the capture to specific instance name parameters. You may use shell “glob” patterns such as *, ?, []. You MUST have TraceFile uncommented as well.

For example, to trace just memory instances with the pattern “sdram”, you would use:

```
TracePattern *sdram*
```

HistoryPattern

HistoryPattern allows you to limit the size of your history file (see above) by limiting the capture to specific instance name parameters. You may use shell “glob” patterns such as *, ?, [].

For example, to record history for just memory instances with the pattern “sdram”, you would use:

```
HistoryPattern *sdram*
```

Step 3: Compressing Simulation Files

Your History and Trace files can be quite large, therefore Denali recommends that you compress these files (along with the SOMA files used during simulation) before attaching them to your E-mail message. For example, you could use the following steps for compressing the files:

```
tar cf mail.tar <historyfile.his> <tracefile.trc>
gzip mail.tar
```

this will result in a “mail.tar.gz” file which you can E-mail to “support@denali.com”.

Step 4: Composing the E-mail Message

Your E-mail to *support@denali.com* should include the following information:

- a succinct description of the problem you are experiencing, including specific error messages and/or time stamp for the unexpected behavior.
- a description of your simulation environment (i.e. simulator, operating system, etc.)
- your contact information (i.e. telephone number and an appropriate return E-mail address)
- and finally, the file *mail.tar.gz* (from Step 2) as an attachment to the E-mail message.

Following this protocol will ensure that your technical question gets addressed quickly and efficiently by the Denali support staff.

A.1 The Denali History File

The Denali history file contains very important debug information regarding the Denali memory models. The history file will decode all bus transactions and record all memory events. You can elect to capture more debug information in your history file if so desired. This is explained below.

A.1.1 Understanding the Denali History Files (HistoryFile in .denalirc):

If you have elected to generate the default history file by setting on the **HistoryFile** parameter in your *.denalirc* file, then you will see basic memory read and write operations plus decoded memory commands. An example of this history file is shown below:

Instance	Time	Action	Address	Value
testbench.uut1		LOAD FILE: init2.dat		
testbench.uut1		LOAD FILE: done		
testbench.uut1		LOAD FILE: init.dat		
testbench.uut1		LOAD FILE: done		
testbench.uut1	15 ns	Cycle: 1 Command Nop		
testbench.uut1	105 ns	Cycle: 4 Command Precharge All		
testbench.uut1	135 ns	Cycle: 5 Command Mode Register Set		
testbench.uut1	135 ns	Setting Burst Length = 8 Cas Latency = 2 interleave		
testbench.uut1	165 ns	Cycle: 6 Bank 1 Command Activate		
testbench.uut1	195 ns	Cycle: 7 Command Nop		
testbench.uut1	315 ns	Cycle: 11 Bank 0 Command Activate		
testbench.uut1	345 ns	Cycle: 12 Command Nop		
testbench.uut1	465 ns	Cycle: 16 Bank 0 Command Write		
testbench.uut1	465 ns	MASKED SIM WRITE	000040+0	000 (1FF)
testbench.uut1	495 ns	Cycle: 17 Command Nop		
testbench.uut1	495 ns	MASKED SIM WRITE	000041+0	001 (1FF)
testbench.uut1	525 ns	MASKED SIM WRITE	000042+0	002 (1FF)
testbench.uut1	555 ns	MASKED SIM WRITE	000043+0	003 (1FF)
testbench.uut1	585 ns	MASKED SIM WRITE	000044+0	004 (1FF)
testbench.uut1	615 ns	MASKED SIM WRITE	000045+0	005 (1FF)
testbench.uut1	645 ns	MASKED SIM WRITE	000046+0	006 (1FF)
testbench.uut1	675 ns	MASKED SIM WRITE	000047+0	007 (1FF)
testbench.uut1	795 ns	Cycle: 27 Bank 0 Command Read		
testbench.uut1	825 ns	Cycle: 28 Command Nop		
testbench.uut1	825 ns	SIM READ	000040+0	000
testbench.uut1	855 ns	SIM READ	000041+0	001
testbench.uut1	885 ns	SIM READ	000042+0	002
testbench.uut1	915 ns	SIM READ	000043+0	003
testbench.uut1	945 ns	SIM READ	000044+0	004
testbench.uut1	975 ns	SIM READ	000045+0	005
testbench.uut1	1005 ns	Cycle: 34 Command Burst Stop		
testbench.uut1	1005 ns	SIM READ	000046+0	006

A.1.2 HistoryDebug Mode (HistoryFile AND HistoryDebug in .denalirc):

If you have turned on the HistoryDebug option in the *.denalirc* file you will get additional debug information. This can be helpful in determining the internal settings for certain memory devices. A DRAM example history file with debug is shown below:

Instance	Time	Action	Address	Value
testbench.uut1		LOAD FILE: init2.dat		
testbench.uut1		LOAD FILE: done		
testbench.uut1		LOAD FILE: init.dat		
testbench.uut1		LOAD FILE: done		
testbench.uut1	15 ns	Debug: Cycle 1: State Bank 0: idle 1: idle		
testbench.uut1	15 ns	Cycle: 1 Command Nop		
testbench.uut1	45 ns	Debug: Cycle 2: State Bank 0: idle 1: idle		
testbench.uut1	75 ns	Debug: Cycle 3: State Bank 0: idle 1: idle		
testbench.uut1	105 ns	Debug: Cycle 4: State Bank 0: idle 1: idle		
testbench.uut1	105 ns	Cycle: 4 Command Precharge All		
testbench.uut1	135 ns	Debug: Cycle 5: State Bank 0: idle 1: idle		
testbench.uut1	135 ns	Cycle: 5 Command Mode Register Set		
testbench.uut1	135 ns	Setting Burst Length = 8 Cas Latency = 2 interleave		
testbench.uut1	165 ns	Debug: Cycle 6: State Bank 0: Mode Register Access 1: Mode Register Access		
testbench.uut1	165 ns	Cycle: 6 Bank 1 Command Activate		

```

testbench.uut1 195 ns Debug: Cycle 7: State Bank 0: idle 1: active
testbench.uut1 195 ns Cycle: 7 Command Nop
testbench.uut1 225 ns Debug: Cycle 8: State Bank 0: idle 1: active
testbench.uut1 255 ns Debug: Cycle 9: State Bank 0: idle 1: active
testbench.uut1 285 ns Debug: Cycle 10: State Bank 0: idle 1: active
testbench.uut1 315 ns Debug: Cycle 11: State Bank 0: idle 1: active
testbench.uut1 315 ns Cycle: 11 Bank 0 Command Activate
testbench.uut1 345 ns Debug: Cycle 12: State Bank 0: active 1: active
testbench.uut1 345 ns Cycle: 12 Command Nop
testbench.uut1 375 ns Debug: Cycle 13: State Bank 0: active 1: active
testbench.uut1 405 ns Debug: Cycle 14: State Bank 0: active 1: active
testbench.uut1 435 ns Debug: Cycle 15: State Bank 0: active 1: active
testbench.uut1 465 ns Debug: Cycle 16: State Bank 0: active 1: active
testbench.uut1 465 ns Cycle: 16 Bank 0 Command Write
testbench.uut1 465 ns Debug: Write (0, 000, 040)
testbench.uut1 465 ns MASKED SIM WRITE 000040+0 000 (1FF)
testbench.uut1 495 ns Debug: Cycle 17: State Bank 0: active 1: active
testbench.uut1 495 ns Cycle: 17 Command Nop
testbench.uut1 495 ns Debug: Write (0, 000, 041)
testbench.uut1 495 ns MASKED SIM WRITE 000041+0 001 (1FF)
testbench.uut1 525 ns Debug: Cycle 18: State Bank 0: active 1: active
testbench.uut1 525 ns Debug: Write (0, 000, 042)
testbench.uut1 525 ns MASKED SIM WRITE 000042+0 002 (1FF)
testbench.uut1 555 ns Debug: Cycle 19: State Bank 0: active 1: active
testbench.uut1 555 ns Debug: Write (0, 000, 043)
testbench.uut1 555 ns MASKED SIM WRITE 000043+0 003 (1FF)
testbench.uut1 585 ns Debug: Cycle 20: State Bank 0: active 1: active
testbench.uut1 585 ns Debug: Write (0, 000, 044)
testbench.uut1 585 ns MASKED SIM WRITE 000044+0 004 (1FF)
testbench.uut1 615 ns Debug: Cycle 21: State Bank 0: active 1: active
testbench.uut1 615 ns Debug: Write (0, 000, 045)
testbench.uut1 615 ns MASKED SIM WRITE 000045+0 005 (1FF)
testbench.uut1 645 ns Debug: Cycle 22: State Bank 0: active 1: active
testbench.uut1 645 ns Debug: Write (0, 000, 046)
testbench.uut1 645 ns MASKED SIM WRITE 000046+0 006 (1FF)
testbench.uut1 675 ns Debug: Cycle 23: State Bank 0: active 1: active
testbench.uut1 675 ns Debug: Write (0, 000, 047)
testbench.uut1 675 ns MASKED SIM WRITE 000047+0 007 (1FF)
testbench.uut1 705 ns Debug: Cycle 24: State Bank 0: active 1: active
testbench.uut1 735 ns Debug: Cycle 25: State Bank 0: active 1: active
testbench.uut1 765 ns Debug: Cycle 26: State Bank 0: active 1: active
testbench.uut1 795 ns Debug: Cycle 27: State Bank 0: active 1: active
testbench.uut1 795 ns Cycle: 27 Bank 0 Command Read
testbench.uut1 825 ns Debug: Cycle 28: State Bank 0: active 1: active
testbench.uut1 825 ns Cycle: 28 Command Nop
testbench.uut1 825 ns Debug: Read (0, 000, 040)
testbench.uut1 825 ns SIM READ 000040+0 000
testbench.uut1 855 ns Debug: Cycle 29: State Bank 0: active 1: active
testbench.uut1 855 ns Debug: Read (0, 000, 041)
testbench.uut1 855 ns SIM READ 000041+0 001
testbench.uut1 885 ns Debug: Cycle 30: State Bank 0: active 1: active
testbench.uut1 885 ns Debug: Read (0, 000, 042)
testbench.uut1 885 ns SIM READ 000042+0 002
testbench.uut1 915 ns Debug: Cycle 31: State Bank 0: active 1: active
testbench.uut1 915 ns Debug: Read (0, 000, 043)
testbench.uut1 915 ns SIM READ 000043+0 003
testbench.uut1 945 ns Debug: Cycle 32: State Bank 0: active 1: active
testbench.uut1 945 ns Debug: Read (0, 000, 044)
testbench.uut1 945 ns SIM READ 000044+0 004
testbench.uut1 975 ns Debug: Cycle 33: State Bank 0: active 1: active
testbench.uut1 975 ns Debug: Read (0, 000, 045)
testbench.uut1 975 ns SIM READ 000045+0 005
testbench.uut1 1005 ns Debug: Cycle 34: State Bank 0: active 1: active
testbench.uut1 1005 ns Cycle: 34 Command Burst Stop
testbench.uut1 1005 ns Debug: Read (0, 000, 046)
testbench.uut1 1005 ns SIM READ 000046+0 006

```

These additional debug comments will provide valuable information such as:

1. The state of the banks (for DRAM devices):

```
testbench.uut1 945 ns Debug: Cycle 32: State Bank 0: active 1: active
```

2. The decoded Bank, Row, and Column addresses used to decode the DRAM:

```
testbench.uut1 975 ns Debug: Read (0, 000, 045)
```

In this case, the Bank address=0x0, the Row Address=0x000 and the Column address=0x045 for this read operation.

A.2 Understanding the History File

There are a few transactions in the history file that need to be described. The Debug entries shown below will only occur in the history file if **HistoryDebug** is turned ON.

A.2.1 SIM READ Entry

Instance Name	Time	Action	Address+Offset	Read Data
testbench.uut1	25095 ns	SIM READ	000060+0	024

A.2.2 MASKED SIM Write Entry

Instance Name (Mask)	Time	Action	Address+Offset	Write Data
testbench.x (3FFFF)	202500 ps	MASKED SIM WRITE	00002+0	0000C

A.2.3 Debug Read

Instance Name	Time	Debug	Action (Bank, Row, Column Addresses)
testbench.uut1	945 ns	Debug:	Read (0, 000, 044)

A.2.4 Debug Write

Instance Name	Time	Debug	Action (Bank, Row, Column Addresses)
testbench.uut1	615 ns	Debug:	Write (0, 000, 045)

A.2.5 File Load

Instance Name	Action (VIRT="logical" memory) : Filename/done
testbench.uut1	LOAD FILE: init.dat
testbench.uut1	LOAD FILE: done
32-Bit-Data	VIRT LOAD FILE: mem32x8.dat
32-Bit-Data	VIRT LOAD FILE: done

Symbols

.denalirc 54

A

Address 61
Address Scrambling 96, 105
addressing scheme 61
Assertion Messages 46
AssertionMessages 47, 53
Assertions 82

B

Bit-blast Pins 22

C

Callback 108
Callback Interface 108
Changing Timing Parameters 72
Clock Cycle Recalculation 76
ClockStableCycles 44, 53
Close Window 35
Contents Table Origin 35
Conventions, typographical 7
coupling faults 90
Creating memory faults 90
Customer support 8

D

Data access assertions 83
Data Bit reordering and masking 96
DDR-II 54
DDR-II SDRAM specific .denalirc parameters 48
Debug 35
Denali technical support 9
DenaliMemCallback 110
denaliMemCallback 109, 110, 111
DenaliOwn 37, 54
DenaliOwnClass 37, 54
denaliPcieCallback 108
denalirc 39
Depth Expansion 93, 94
depth expansion 101
DifferentialClockChecks 46, 53
DifferentialClockSkew 46
DifferentialClockSkewClock Skew 46
DRAM 12
Dynamically Enabling and Disabling Assertions 86

E

eDRAM 53
EiMessages 46, 53
Embedded ASIC/FPGA 12
eMemory 14
enableReason 110
Enabling Error Injection on “backdoor” Reads 89
Enabling/Disabling Fault Checks 91
Error Count Variable 45
Error Injection 87
Error Message Control 75
ErrorCount 53
ErrorMessages 45, 53
ErrorMessagesStartTime 45, 53
eSSRAM 54
ESSRAM specific .denalirc parameters 49

Event 110
ExitOnErrorCount 45, 53

F

FAQs 8
Fault Modeling 90

G

getCallback() 111
Global memory access assertions 85

H

HDL Shell 23
History File 197
HistoryDebug 40, 52, 196
HistoryDebugLoad 40, 52
HistoryFile 39, 52, 196
HistoryInSimLog 40, 52
HistoryPattern 43, 52

I

IBM-EDRAM Specific .denalirc parameters 47
InitChecks 53
InitChecksPauseTime 46, 53
Initialization Checks 44
Initializing Memories 61
InitialMemoryValue 42, 52
InitMessages 43, 52
InitMrsAddressStable 48, 54
Interleaving 95
interleaving 101
Interleaving Example 95
IrregularClock 43, 53

L

License Checkout 55
LicenseQueueRetryDelay 41, 52
LicenseQueueTimeout 41, 52
Licensing Solutions 55
Loading memories from a file 63
Logical Addressing with MMAV 92

M

Masked Memory Writes 70
masking 101
Memory Access Assertions 82
Memory Address Determination 61
Memory Content File Format 64
Memory Contents Window 34
Memory Holes 97
Mentor Graphics ModelSim specific .denalirc parameters 49
Mentor Seamless 113
Mentor Seamless HW/SW Co-Verification specific .denalirc parameters 50
mmassert_access 83
MMAV 39
mmbreak 59
mmcomp 71
mmCreateScratchpad 107
mmcreatescratchpad 106, 108
mmdebugoff 75
mmdebugon 75
mmdisablecallback 60
mmdisablecallbackall 59
mmenablecallback 59

- mmerrinject 88
- mmerrormessagesoff 75
- mmerrormessageson 75
- mmexit 59
- mmfault 90
- mmgetids 58
- mmgetinfobyid 58
- mmgetinstanceid 66, 67
- mminstanceid 66
- mmload 63
- mmnote 58
- mmreadword 68
- mmreadword2 68
- mmreadword3 68
- mmrecalculatecycle 72
- mmreset 67
- mmsave 71
- mmsaverange 71
- mmsetaccesscallbackmask 60
- mmsetallerrinject 88
- mmsetallfault 91
- mmsetfault 91
- mmSetFillValue 62
- mmsetvar 55
- mmsimulationdatabase 31
- mmsomaload 74
- mmsomaset 73, 74
- mmstartpureview 32
- mmTclCallback 57
- mmTclEval 56
- mmtcleval 31, 55, 72
- mmtime 59
- mmwriteword 69
- mmwriteword2 69
- mmwriteword3 69
- mmwriteword4 69
- mmwriteword5 69
- mmwritewordmasked 70
- ModelSim 54
- ModelSimTimeDefinitionToggle 48, 49, 50, 51, 54

N

- NC 119
- NCSIM 56
- Non-Volatile/Flash Memories 12
- noXsInReadData 49

O

- OffChipDriveImpedanceChecks 54
- Open Instance 35
- Open Simulation Results 32
- Output Timing 44
- OutputTiming 53

P

- Parity Bits-bit position in memories 65
- Parity Checking Assertions 86
- PureSpec
 - help and documentation 8
- PureView 17, 31

R

- RAMBUS 76
- RAMBUS - .denalirc Options 78

- RAMBUS - Channel Delay Settings 77
- RAMBUS - Device id's 77
- RAMBUS - Device Numbers 77
- RAMBUS - Device Refresh Options 78
- RAMBUS - Multi-Bank Refresh Options 79
- RAMBUS - Turbo Mode 79
- RandomInsteadOfXInReadData 49
- RandomOutputDelay 44, 53
- RDRAM - Byte Ordering Options 79
- RDRAM (Rambus) Specific Options 45
- RDRAM specific .denalirc parameters 47
- ReadDQSContentionCheck 42, 52
- Reading and Writing Memories 68
- reason 111
- Recalculating Clock Cycle 72
- Refresh 35
- RefreshChecks 42, 52
- RefreshOnReadWrite 42, 52
- Register File Specific .denalirc parameters 47
- Re-loading SOMA Files 72
- RLDRAM 54
- RLDRAM specific .denalirc parameters 48
- RldramInitCyclesCheck 48, 54
- RldramInitRefreshChecks 48
- Run Tcl Command 35

S

- Save SOMA 20
- Save SOMA As 20
- Save Source 20
- Save Source As 20
- Saving and Comparing Memory Contents 70
- Scratchpad Memories 106
- Seamless 54
- setCallback 110
- Show Last Read 35
- Show Last Write 35
- Simulation
 - Vera interface 166
- SimulationDatabase 37, 41, 52
- SimulationDatabaseBuffering 41, 52
- SimulationDatabasePattern 41, 52
- Simulator Queuing 55
- SOMA 12
- SOMA Output Format 21
- SRAM 12
- STD_ULOGIC for VHDL Ports 22
- stuck-at faults 90
- SuppressPortContention 54
- SuppressRefreshInfoMessages 53
- SuppressUnknownAddrReadError 47

T

- Tcl 56
- Tcl Callback Helper Commands 58
- Tcl Interface 56
- Tcl with ModelSim 56
- Tcl with NCSIM 56
- TclInterp 45, 46, 53
- Technical Support 196
- Technical support 9
- TimingChecks 41, 52
- TimingChecksReportOnly 48, 78
- TimingChecksStartTime 48, 78

- Trace Backdoor Reads/Writes 47
- TraceBackdoorReadWrite 53
- TraceBackdoorReadWrite 0 47
- TraceFile 40, 52, 196
- TracePattern 43, 52
- TraceTimingChecks 40, 52
- TrackAccessFromInit 46, 53
- Transaction History 36
- Transaction Summary 36
- transGet 110
- transId 109, 111
- transition faults 90
- Typographical conventions 7

U

- Using 11

V

- Value Format 35
- Vera interface
 - overview 142, 169
 - simulation 166
- Verilog Callbacks 108

W

- WarnSuppress 47
- Width Expansion 93
- width expansion 101

X

- XML Basics 92